THE UNIVERSITY *of* EDINBURGH

Title	Compiler cost model for speculative multithreading chip-multiprocessor architectures
Author	Dou, Jialin.
Qualification	PhD
Year	2006

Digitisation notes:

- Page number 2, 6, 16, 18, 22, 40, 52, 54, 92, 104, 106, 108 is skipped in original text.

A Compiler Cost Model for Speculative Multithreading Chip-Multiprocessor Architectures

Jialin Dou

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2006

Abstract

Speculative parallelisation (also known as speculative multithreading and thread-level speculation) is a technique that complements automatic compiler parallelisation by allowing code sections that cannot be fully analysed by the compiler to be aggressively executed in parallel. However, while speculative parallelisation can potentially deliver significant speedups, several overheads associated with this technique can limit these speedups in practice.

This thesis proposes a novel compiler static cost model of speculative multithreaded execution that can be used to predict the resulting performance. This model attempts to predict the expected speedups, or slowdowns, of the candidate speculative sections based on the estimation of the combined run-time effects of various speculation overheads, and taking into account the scheduling restrictions of most speculative multithreading execution environments. The model is based on estimating the likely execution duration of threads and considers all the possible permutations of these threads when scheduled on a multiprocessor. Also, different from heuristics that attempt to qualitatively estimate potentially "good" or "bad" sections for speculative multithreaded execution, this model allows the compiler to estimate the speedup or slowdown quantitatively. Such quantitative estimate can then be used by the compiler or run-time system to make more complex and educated tradeoff decisions.

The proposed cost model was implemented in a research compiler development framework. The model seamlessly uses the compiler's intermediate representation and integrates with the control and data flow analyses. The resulting framework was tested and evaluated on a collection of SPEC benchmarks, which include large real-world scientific and engineering applications. The framework was found to be very stable and efficient with moderate compilation times.

Initially, the proposed framework is evaluated on a number of loops that suffer mainly from load imbalance and thread dispatch and commit overheads. Experimental results show that the framework can identify on average 68% of the loops that cause slowdowns and on average 97% of the loops that lead to speedups. In fact, the framework predicts the speedups or slowdowns with an error of less than 20% for an average of 44% of the loops across the benchmarks, and with an error of less than 50% for an average of 84% of the loops. Overall, the framework leads to a performance improvement of 5% on average, and as high as 38%, over a naive approach that attempts to

speculatively parallelise all the loops considered.

The proposed framework is also evaluated on loops that may suffer from data dependence violations. Experimental results with all loops show that prediction accuracy is lower when loops with violations are included. Nevertheless, accuracy is still very high for a static model: the framework can identify on average 45% of the loops that cause slowdowns and on average 96% of the loops that lead to speedups; it predicts the speedups or slowdowns with an error of less than 20% for an average of 28% of the loops across the benchmarks, and with an error of less than 50% for an average of 80% of the loops. Overall, the framework often outperforms, by as much as 25%, a naive approach that attempts to speculatively parallelize all the loops considered, and is able to curb the large slowdowns caused in many cases by this naive approach.

Acknowledgements

First of all, I would like to thank my supervisor, Dr. Marcelo Cintra, for his kind support and help throughout all my study, for teaching me how to do genuine research, for making me confident solving even the most frustrating problem, and for making my thinking more insightful, creative and systematical. I want to thank my second supervisor, Dr. Michael O' Boyle for his critical advices during my study.

I would also like to thank everyone in our research group for sharing our common interests and discussions. I want to thank my officemate Worawan Maruringsith for our discussions and for her help during my study.

I would like to thank my parents for their warm support throughout all my time in the UK. Finally, I want to thank my girlfriend Lu Zhou for making my personal life in the UK so enjoyable.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following paper:

- Jialin Dou and Marcelo Cintra. "Compiler Estimation of Load Imbalance Overhead in Speculative Parallelization." *Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 203-214, September 2004.

Jialin Dou

Table of Contents

List of Figures

List of Tables

Part I

Introduction and Background

Chapter 1

Introduction

1.1 Motivation

As exploitation of greater degrees of instruction level parallelism seems to provide diminishing gains [1, 34], thread level parallelism becomes a more attractive choice. It offers parallelism in a more coarse level, such as loop iterations or procedure bodies. On the other hand, with advances in fabrication technology, there is a trend toward chip multiprocessor (CMP) systems. They offer better power efficiency, by delivering the same, or better, peak performance as a superscalar processor of comparable size, while enabling each individual processor to run at a lower clock frequency. However, to utilise the computing power given by CMP's, it is necessary to partition the program so that it can run in parallel on multiprocessors. The most common approach now is by using explicit parallel programming, though it is not yet commonplace and requires a huge amount of well-trained programmer effort. Another approach is to use automatic parallelising compilers[4, 18]. However, currently the technology is still not mature, and these still fail to parallelise a significant set of codes when data dependence information at compile time is incomplete, for example, in the presence of a pointer alias, or a subscripted subscript.

To aid in the parallelisation process, hardware support for *speculative parallelisation* (also known as *thread-level speculation* or *speculative multithreading*) has been proposed [2, 15, 19, 23, 27, 41, 42, 46]. In this approach, potentially dependent threads are speculatively executed in parallel and hardware mechanisms monitor the memory reference stream for any data dependence violation. If a dependence violation is de-

tected, the system reverts back to a safe state and threads are re-executed.

While speculative parallelisation can potentially deliver significant speedups for code sections that would otherwise be executed sequentially, several overheads associated with the technique limit these speedups in practise. In fact, in many cases these overheads can lead to slowdowns with respect to a sequential execution. Thus, accurately identifying, and quantifying, these overheads is critical for good overall performance. Five major sources of overheads in speculative parallelisation have been identified [32, 33, 48, 49]: thread *squash and restart* due to data dependence violations, speculative *buffer overflow*, *load imbalance*, thread *dispatch and commit*, and inter-thread *communication*.

Current compiler technology for speculative parallelisation is still maturing. Early compiler analyses to select appropriate speculative threads were based on simple heuristics and only indirectly estimate the speculative multithreaded execution overheads [5, 8, 21, 24, 49, 53]. Also, these heuristics tackle only a subset of the types of overheads and there is no integrated approach to consider all overheads jointly. Finally, these heuristics only provide a qualitative prediction of the suitability of code sections for speculative multithreaded execution.

1.2 Contributions of This Thesis

Firstly, this thesis proposed and evaluated a method to quantitatively evaluate the load imbalance overhead under speculative parallelisation. To the best of my knowledge, this is the first compiler technique that enables the quantitative estimate of load imbalance overhead under speculative parallelisation.

Secondly, this thesis proposed and evaluated a static compiler cost model to quantitatively estimate the major speculation overheads incurred by the execution model of speculative parallelisation. To the best of my knowledge, this is the first compiler model that enables the estimation of the joint effect of all the major speculation overheads on speculative parallelisation performance, and it is the first compiler model that provides an estimate of the speedup, which is a direct estimate of the benefit of speculative parallelisation. This knowledge can assist the compiler or run-time system to make more complex and educated tradeoff decisions. For instance, in a highly loaded multiprogrammed environment the compiler or run-time system may decide to switch

off speculative parallelisation even when a speedup is expected if this speedup is too small and does not justify the use of the extra resources.

1.3 Thesis Structure

This thesis is structured as four parts. Part I has three chapters, in which, Chapter 2 describes the background information about the execution model of speculative multi-threaded execution and the related speculation overheads and Chapter 3 discusses the related work.

Part II describes the base tuple model and how it is applied to measure the load imbalance overhead. This part is composed of Chapter 4, Chapter 5 and Chapter 6. In particular, Chapter 4 provides a detailed discussion of load imbalance overhead and Chapter 5 presents the base tuple model and the data structures for the model and provides an example computation using the base tuple model. The evaluation of the model is contained in Chapter 6.

Part III describes the extended tuple model and how it is used to estimate the other overheads. This part is composed of Chapter 7, Chapter 8 and Chapter 9. Chapter 7 provides a detailed explanation of the speculation overheads and then introduces the extended tuple model and explains how the other speculation overheads are incorporated into the model. The algorithm for the model and an example computation are also provided in this chapter. The evaluation of the extended tuple model is contained in Chapter 8. The Chapter **??** summaries all the simplifications made in the thesis and discussed their effects on the prediction accuracy.

Finally, Part IV concludes this thesis. In this part, an outlook for future work is given in Chapter 10 and Chapter 11 summarises this thesis.

Chapter 2

Background

2.1 Speculative Execution

2.1.1 Execution Model

Under the *speculative parallelisation* (also called *thread-level speculation* or *speculative multithreading*) approach, sequential sections of code are speculatively executed in parallel hoping not to violate any sequential semantics. The control flow of the sequential code imposes a total order on the threads. At any time during execution, the earliest thread in program order is *non-speculative* while the others are *speculative*. The terms *predecessor* and *successor* are used to relate threads in this total order. Stores from speculative threads generate unsafe *versions* of variables that are stored in some sort of *speculative buffer*, which can be simply the private caches or some additional storage dedicated to speculative parallelisation. If a speculative thread overflows its speculative buffer it must stall and wait to become non-speculative. Loads from speculative threads are provided with potentially incorrect versions. As execution proceeds, the system tracks memory references to identify any cross-thread data dependence violation. If a dependence violation is found, the offending thread must be *squashed*, along with its successors, thus reverting the state back to a safe position from which threads can be re-executed. In addition to this implicit memory communication mechanism, some hardware environments for speculative parallelisation allow synchronised *memory and register communication* between neighbour threads. When the execution of a non-speculative thread completes it *commits* and the values it generated can be moved to safe storage (usually main memory or some shared lower-level cache). At this point

its immediate successor acquires non-speculative status and is allowed to commit. If a speculative thread completes it must wait for all predecessors to commit before it can commit. After committing, the processor is free to start executing a new speculative thread. Usually a processor that completes the execution of a speculative thread before the predecessor threads have committed is not allowed to start execution of a new speculative thread.

Figure 2.1 shows an example loop under speculative parallelisation. The parallelising compiler fails to parallelise this loop because the data dependence between the load and the store operations to array *A* is uncertain at compile time. This loop is selected to be parallelised speculatively. However, during run time, a data dependence between iteration J+2 and iteration J is detected, so the threads are squashed and restarted again.

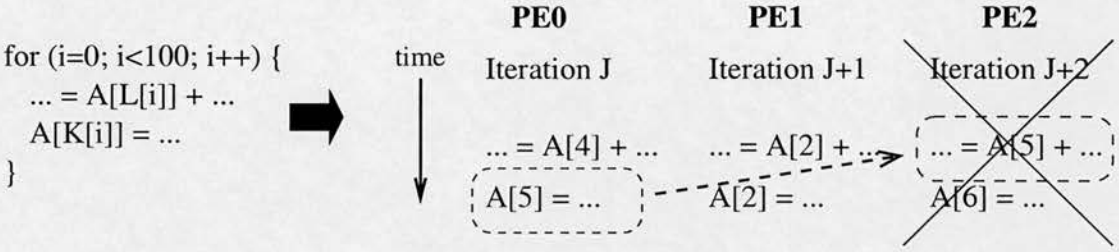

Figure 2.1: The threads are squashed because after the store operation to A[5] is detected in iteration J, the earlier load operation to A[5] from iteration J+2 becomes invalid.

As loops iterations can be easily split into threads and they account for a large fraction of the program execution time for many classes of applications, the compiler model proposed in this thesis assumes loop iterations as the sources of speculative threads. Other sources of speculative threads that have been proposed in previous speculative multithreading proposals include control branches and procedure calls. The research for the cost models for such sources is out of the scope of this thesis.

2.1.2 Speculative Parallelisation Overheads

The execution model of speculative parallelization (Section 2.1.1) leads to five major overheads: thread *squash and restart* due to data dependence violations, speculative *buffer overflow*, *load imbalance*, thread *dispatch and commit*, and inter-thread *communication*. The thread *squash and restart* overhead is mainly composed by the time to

flush the speculative buffers and the redundant re-execution of the part of the thread prior to the violation. Figure 2.2b depicts this overhead. This overhead is dictated by the actual frequency of data dependence violations and by the location of the dependences within the threads (e.g., the actual location of the store within thread 1 in Figure 2.2b determines how much work has to be redone in threads 3 and 4). The speculative *buffer overflow* overhead relates to the amount of time the processor remains idle after a speculative thread overflows its speculative buffer and until it is allowed to proceed. Figure 2.2c depicts this overhead. This overhead is dependent on the physical size and organization of the speculative buffer, the amount of data written by the thread, and the size of threads (e.g., the amount of data stored determines the location of the overflowing store in thread 2 in Figure 2.2c and, thus, the amount of work that must be completed once the thread is allowed to proceed). The thread *dispatch and commit* overhead is mainly composed by the time to move the speculatively modified data from the speculative buffer to safe storage and the time to update the system state to reflect the new status of the threads. This overhead depends mainly on the amount of data written by the thread. The inter-thread *communication* overhead relates to the time the processor remains idle while a speculative thread waits for a memory or register value produced and communicated by a predecessor thread. This overhead is only relevant in architectures that support synchronized register or memory communication and is dependent on the frequency of such communication points and the location of these within the threads. Finally, *load imbalance* overhead relates to the time the processor remains idle after completing the execution of a speculative thread and until this thread becomes non-speculative. Figures 2.2a and 2.2c depict this overhead. This overhead depends mainly on the differences in execution time among threads, which is in turn heavily influenced by the other overheads above (e.g., the smaller execution time of thread 3 compared to thread 2 causes the load imbalance in processor 2 in Figure 2.2a; and the overflow of thread 2 causes further load imbalance in processors 2 and 3 in Figure 2.2c). Of the five overheads discussed above, thread squash and restart, speculative buffer overflow, and load imbalance have been identified as the most significant overheads [32, 48, 49].

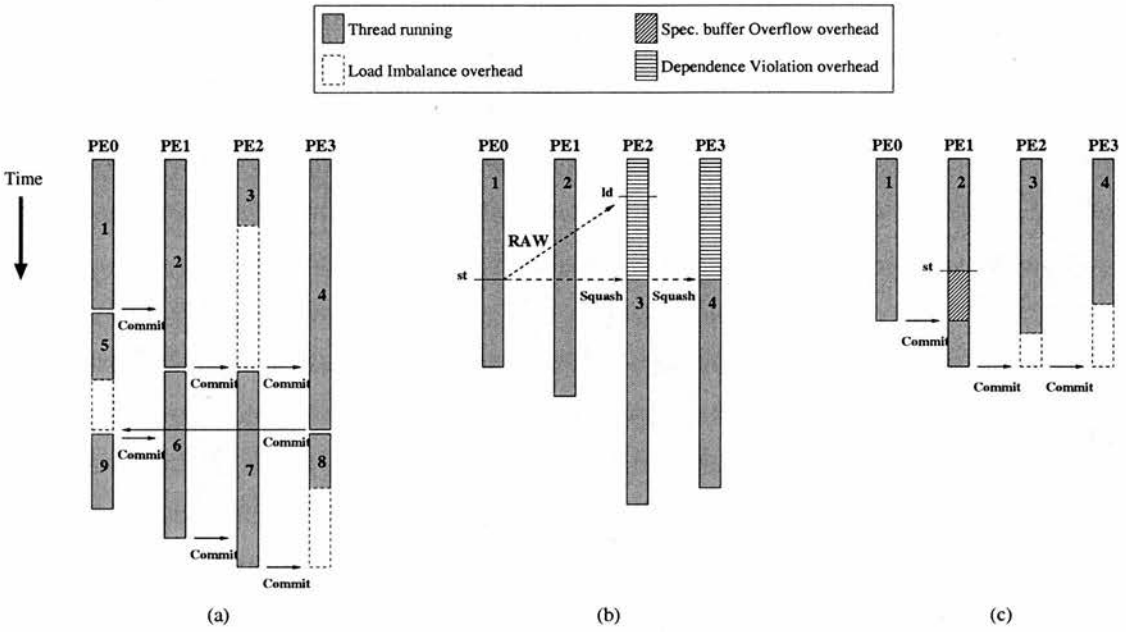

Figure 2.2: Some speculative parallelisation overheads: (a): load imbalance; (b): squash and restart; and (c) speculative buffer overflow, also leading to further load imbalance overhead. The numbers inside the bars correspond to the order in which the threads are scheduled.

Chapter 3

Related Work

3.1 Speculative Multithreading Architecture Schemes

Architectural support for speculative parallelisation in CMP or multithreaded processors has been extensively investigated. There have been two major approaches for exploring speculative thread-level parallelism on a CMP.

In the first approach, the CMP is very generic and requires minimal additional hardware support beyond that of a traditional SMP. Such proposals tend to restrict threads to communicating via the memory. They also are typically used to exploit only speculative loop-level parallelism and require additional compiler support that is aware of the underlying speculative multithreading architecture. The compiler framework proposed in this thesis assumes such an underlying architecture approach. Typical architecture proposals in this category include the Stanford Hydra CMP [20], the Stampede architecture [42, 45], and the Superthreaded Processor Architecture [46].

In the second architecture approach, the CMP is highly customised to fully explore speculative parallelisation and requires minimal compiler support. Such proposals typically support register communications and, thus, the processors are connected in a ring or using switches. Since the processors are tightly coupled, such proposals are generally tuned to explore finer grained parallelism than the first approach. Representative architecture proposals are The Multiscalar Processor [41], the Clustered Speculative Multithreaded Processor [28], and the Multiplex Processor [32].

There are also early works that try to explore speculative multithreading on a single processor. Such proposals include the Speculative Multithreaded Processor [27],

the Dynamic Multithreading Processor [2] and the Single-Program Speculative Multithreading (SPSM) Architecture [15]. Such proposals are similar to an SMT [1] approach but with speculative multithreading support, which enables the processors to break the bottleneck of the control and data flows by aggressively extracting threads from the single program far beyond the current instruction window.

Architectural support for speculative parallelisation in scalable multi-chip multiprocessors has also been investigated [9, 43, 54]. Some of the above works have identified and measured the main overheads of speculative parallelisation. Additional hardware-based support to reduce some of the overheads of speculative parallelisation have also been investigated [10, 16, 30, 37, 44], but such support tend to be costly. Alternatively to hardware-assisted speculative parallelisation, software-only approaches have also been proposed [11, 12, 39].

3.2 Compiler Support for Speculative Multithreading

Compiler technology for speculative parallelisation is still a maturing field and most current techniques are based on heuristics or simple analyses.

[48, 49]: is one of the first compiler works on selection of speculative threads. In this work, major overheads that impair the speculative parallelisation performance were identified. The techniques proposed were based on heuristics based on threshold of thread size, addressed the importance of avoiding threads with inter-threads data dependence violations, and avoiding such violations by avoiding spawning threads with inter-thread dependence. This work, however, did not give a quantitative estimation of the violation overhead. It differs from our work in that it addressed the thread partitioning from any program region and not only loop iterations, i.e., basic blocks. So this work also addressed other issues, such as control flow dependence, and how to partition threads to minimise control flow mis-speculation. This is because this work assumed the Multiscalar architecture as the underlying speculative multithreading framework, which is good at exploiting finer-grained parallelism, since it supports register communications between processors. This work used a simple heuristic that tries to amortise the overheads by considering only threads with estimated sizes within a certain range.

[1] SMT stand for Simultaneous multithreading, which is a single processor multithreading architecture, which has multiple functional units and allows multiple threads to be alive concurrently to utilise fully utilise the functional units. Typical commencial implementation is the Intel Pentium 4 processor.

[21] described the structure of the "Multiplex compiler". This compiler framework was integrated into the Polaris parallelising compiler, which only has the Fortran frontend. In this compiler, loops are selected as implicit multithreading or explicit multithreading based on the estimation of speculative buffer overflow and the threshold of the loop workload. To identify sources of speculative buffer overflow, this work uses cache miss equations to statically detect potential conflicts. Cache miss equations work well for affine array references, but are not well suited for more irregular access patterns. A limitation of this work is that it only considers the inner loops, while the model proposed in this thesis can handle loops at all nesting levels.

[5] proposed a compiler framework that tries to search for the optimal thread partitioning from a control flow graph by using a count of possible cross-thread dependences. This work assumes that threads are partitioned from any program region, which is different from the thread partitioning scheme used in this thesis, since the cost model proposed in this thesis assumes that threads are only partitioned from loop iterations. The partitioning schemes proposed the above work, try to avoid partitioning threads with data dependences by using a simple heuristic based on the analysis of what was defined as "data dependence count", which is a weighted count of the number of dependence arcs from a basic block to another, and "data dependence distance", which corresponds to the difference of the time stamps between the producer and the consumer memory operations.

[8] used probabilistic dependence analysis to estimate the likelihood of data dependence violations. While the major contribution of this paper is the technique of the probabilistic points-to analysis, this work also provided a simple qualitative model for estimating the violation cost based on the frequency of violations, which was computed based on the results of the probabilistic points-to analysis. The model proposed in this thesis is superior to the cost model in the above work in that it enables the integration of all the major overheads into one analysis model and it provides a quantitative estimation of the costs of the joint effects of various speculation overheads.

[14] proposed a cost-driven compilation framework based on the estimation of the squash and restart overhead of the candidate speculative loops. This work was based on what is called, the "cost graph", which was constructed from the control flow and data dependence graphs. The cost of misspeculation was computed from this graph based on the re-execution probabilities of the threads. Loops are selected to be speculatively

parallelised based on the value of the computed misspeculation cost, the thresholds of loop body sizes and iteration counts, and pre-fork region sizes. Compared with the compiler model proposed in this thesis, the above work only considers squash and restart overheads with some simple heuristics on thread sizes, while the model proposed in this thesis is flexible to include all the major speculation overheads into one framework and is able to predict an overall estimate of speedup. Also, the value of cost in the above work does not indicate whether the loops are going to lead to speedup or slow down, and by how much percent, while the value speedup predicted by the model in this thesis is more intuitive, as the value speedup is a direct indication of the benefit of parallelising a loop.

[38] is perhaps the closest work to the work stated in this thesis. Similar to ours, that framework also analyses all control flow paths and attempts to estimate the performance gains from speculative multithreading. With respect to the execution and cost model, that work differs from ours in that it attempts to emulate a trace-driven-like execution of the threads using the control flow graph while in our work we attempt to model the execution of threads mathematically. Another important difference is with respect to the way threads are generated from the sequential code. While we only consider loop iterations as threads, that work considers threads created from basic blocks between what are called "control quasi-independent points". Finally, unlike ours, that work also considered the possibility, and costs, of adding pre-computation slices to minimize some of the squash and communication overheads.

In addition to the compiler supported thread partitioning and thread selection, previous works have investigated the use of profiling to identify good thread partitioning for speculative execution. [25, 29, 31] use profiling to identify the possible data dependence violations between threads and control flow execution paths to aid the process of thread selection. Alternatively, dynamic partitioning of threads with on-the-fly performance information through hardware monitors has been proposed in [7, 51]. Both approaches usually consider all the overheads combined, but require either feedback-directed or dynamic re-compilation of the code.

3.3 Load Imbalance

Some work has investigated compilation techniques to handle load imbalance in the context of traditional non-speculative parallelisation (e.g., [17, 40]). The techniques proposed in such work target scientific programs with fairly predictable workload distributions and assume flexible scheduling policies that are not comparable to the strict scheduling policies required by speculative parallelisation. To the best of my knowledge, no previous work has focused on the load imbalance problem under speculative parallelisation,

Part II

Thread Tuple Model and Load Imbalance Overhead

Chapter 4

Load Imbalance Overhead

It has been observed that load imbalance can account for a large fraction of the total execution time for 4 and 8 processors in a speculative CMP for SPEC benchmarks [1] [32, 48, 49]. The load imbalance problem is expected to be greater in more irregular applications and in larger speculative CMP configurations. The impact of load imbalance in speculative parallelisation is far greater than in traditional, non-speculative, parallelisation. Figure 4.1 compares the execution under speculative and non-speculative parallelisation. Usually, with speculative parallelisation a processor cannot start work on a new thread until its current thread becomes non-speculative and commits. In non-speculative parallelisation a processor can start work on a new thread as soon as it finishes execution of its current thread.

In addition to not allowing processors to start work on new threads while having pending uncommitted threads, the speculative multithreaded execution model shown in Figure 4.1a is a static thread partitioning scheme, which means that the partitioning of threads happen at compile time [19, 23, 41]. Other execution models are possible with increased software and hardware complexity. In contrast to the static scheme, some speculative multithreaded systems allow threads to be dynamically (run time) forked onto idle processors [2, 15, 27, 32, 42, 46]. Further support to throttle dynamic thread forking based on system load can somewhat alleviate the load imbalance problem [15]. However, the limitation that processors must remain idle after completing execution of a speculative thread still exists and causes load imbalance. Relaxing this limitation

[1] SPEC benchmarks is a standard set of relevant benchmark test programs widely used to evaluate computer system performance. SPEC stands for the Standard Performance Evaluation Corporation. It is a non-profit organisation which maintains the SPEC benchmarks.

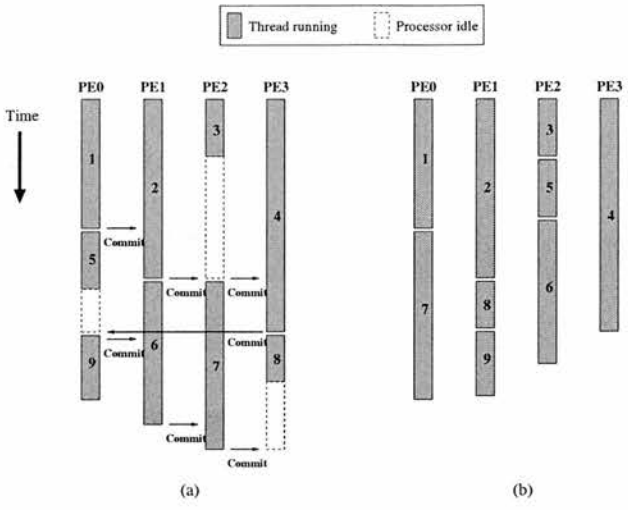

Figure 4.1: Load balancing under (a): speculative parallelisation, threads commit in their original order and new threads can only be assigned to processors after the commit; (b): non-speculative parallelisation, a new thread is assigned to a processor as soon as the current thread finishes. The numbers inside the bars correspond to the order in which the threads are scheduled.

requires costly hardware support to maintain multiple speculative versions of data [9, 16] and, to the best of my knowledge, is not supported by any existing speculative CMP proposal.

Table 4.1 categorises the speculative multithreaded execution models by the thread partition schemes and lists the proposals that support each scheme. In the *static scheduling* scheme, the threads are partitioned by the compiler and scheduled at compile time, while in the *dynamic* scheme the threads are forked from their parent threads at run time by calling thread spawning routines.

Category Criteria	*Value*	*Examples*
Thread partition scheme	Static scheduling	[19, 23, 41]
	Dynamic spawning	[2, 15, 27, 32, 42, 46]

Table 4.1: Category of speculative multithreaded execution by thread partitioning schemes.

The observed load imbalance overhead may indirectly depend on other speculative parallelisation overheads. For instance, the execution delay of a predecessor thread

due to speculative buffer overflow can appear to a successor thread as load imbalance. While a complete compiler estimation of load imbalance overhead is then dependent on the estimation of other overheads, it can be very useful in practise to estimate the load imbalance that is intrinsic to the workload variations across threads. Firstly, previous work indicate that intrinsic variations in thread size due to differences in control flow are in practise a significant fraction of the overall load imbalance overhead [48]. Also, this result can be used as an upper bound on the potential performance gains of speculative parallelisation before analysis of the other overheads. Then, sections of code that are expected to suffer significantly from load imbalance do not have to be considered further. Finally, with an appropriate model of execution such as the one proposed in Chapter 5, this intrinsic load imbalance estimation can be coupled with results from the analyses of other overheads to obtain an overall estimation of expected performance, as shown in Chapter 7.

Workload variations across threads are mainly caused by the following factors: conditional statements, inner loops, and cache misses. In practise, the load imbalance overhead will be amplified by combinations of these factors, such as a conditional statement with an inner loop, or an inner loop with conditional statements.

Chapter 5

Framework

5.1 Basic Idea

Instead of using a collection of heuristics to identify "good" or "bad" code sections for speculative parallelisation, the work presented in this thesis proposes to compute a quantitative estimate of the actual speedup that will be attained. With this result the compiler or run-time system can make an informed decision on whether to try speculative parallelisation or to run those code sections sequentially.

To compute the estimated speedup (S_{est}), a compiler framework of the speculative multithreaded execution was developed from the observation of the run time behaviour of speculative multithreading. This compiler framework is based on modelling the speculative multithreading execution using the estimated thread sizes and probabilities of occurrence of these sizes. The inputs to the model are the number of processors in the system (P), the possible sizes of threads, including overheads, and their probabilities. The model then considers all possible permutations of thread sizes across the P processors. Each permutation, which is defined in this thesis as a *thread tuple*, has a probability of occurrence, which is the joint probability of occurrence of all thread sizes in the tuple. Also, each thread tuple has a sequential and a parallel execution time, which together with the probability of occurrence of the tuple is used to compute the overall estimated sequential ($Tseq_{est}$) and parallel ($Tpar_{est}$) execution times of the average tuple, and, thus, the estimated speedup (S_{est}).

5.2 The Thread Tuple Model

The first step in the model is to compute the possible thread sizes. This is initially done by inspection of the code considering all possible execution paths. Load imbalance that is intrinsic to the code structure due to variable control paths is implicitly handled at this time. Other overheads can then be added as additional thread sizes with their respective probabilities of occurrence. The following describes the model and shows how to compute the base thread sizes due to intrinsic load imbalance. Part III shows how to compute the additional thread sizes generated by the other overheads and how to include them in an extended version of the model.

Assume that based on the possible execution paths and on the other speculative execution overheads there are N possible thread sizes $W_1, ..., W_N$, with probabilities of occurrences $p_1, ..., p_N$ [1]. Further assume, without loss of generality, that these thread sizes form an ordered set A, i.e., $A = \{W_1, W_2, ..., W_N\}$ and $W_1 \leq W_2 \leq ... \leq W_N$. Then there are N^P possible permutations of these thread sizes on P processors, and thus N^P thread tuples. More formally, a thread tuple is an element of A^P, where AxA is the Cartesian product. The probability of occurrence of a particular thread tuple i, which is referred to by p_{tuple_i}, is the product of the probabilities of occurrences of each thread size in the tuple.

As an example, Figure 5.1a shows a loop with two execution paths and Figure 5.1b shows the two possible thread sizes derived from the execution paths. The thread size $W1 = w_1 + w_2 + w_3 + w_4$ has probability $p_1 = p$, where p is the probability of branching to the *then* part of the *if* statement. The thread size $W2 = w_1 + w_5 + w_6$ has probability $p_2 = 1 - p$

[1] Here it is implicitly assumed that the events associated with the parallel threads following some control path are independent, which may not always be true when there is some correlation among certain paths.

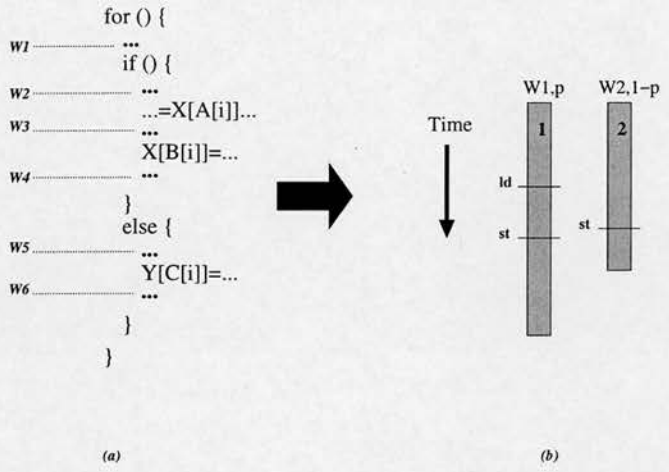

Figure 5.1: An example loop and the thread sizes derived from it.

When running this loop on a four processor speculative multithreading environment, there are $2^4 = 16$ possible permutations of these two thread sizes, where each permutation is called a tuple. This is shown in Figure 5.2.

For a given thread tuple i, the parallel and sequential execution times for the tuple are given by:

$$Tseq_{tuple_i} = \sum_{W_j \in tuple_i} W_j \tag{5.1}$$

$$Tpar_{tuple_i} = \max_{W_j \in tuple_i} W_j \tag{5.2}$$

Equation 5.1 simply indicates that the sequential execution time of the group of threads in a thread tuple is given by the sum of their execution times, while Equation 5.2 indicates that the parallel time of the same group of threads is simply the execution time of the largest of the threads. Note that in this way Equation 5.2 is an approximation to the execution model of Figure 4.1a as it does not take into account the relative position of the largest thread [2].

The thread sizes that are intrinsically derived from the possible execution paths are called the *base thread sizes*. All the the thread tuples formed by the base thread sizes are called *base tuples*. Thus, such *base tuples* do not suffer from overheads of

[2]The relative position of the largest thread is important as processors running its predecessor threads can be assigned one new speculative thread (see Figure 4.1a), thus adding some flexibility to the schedule that is not captured by Equation 5.2.

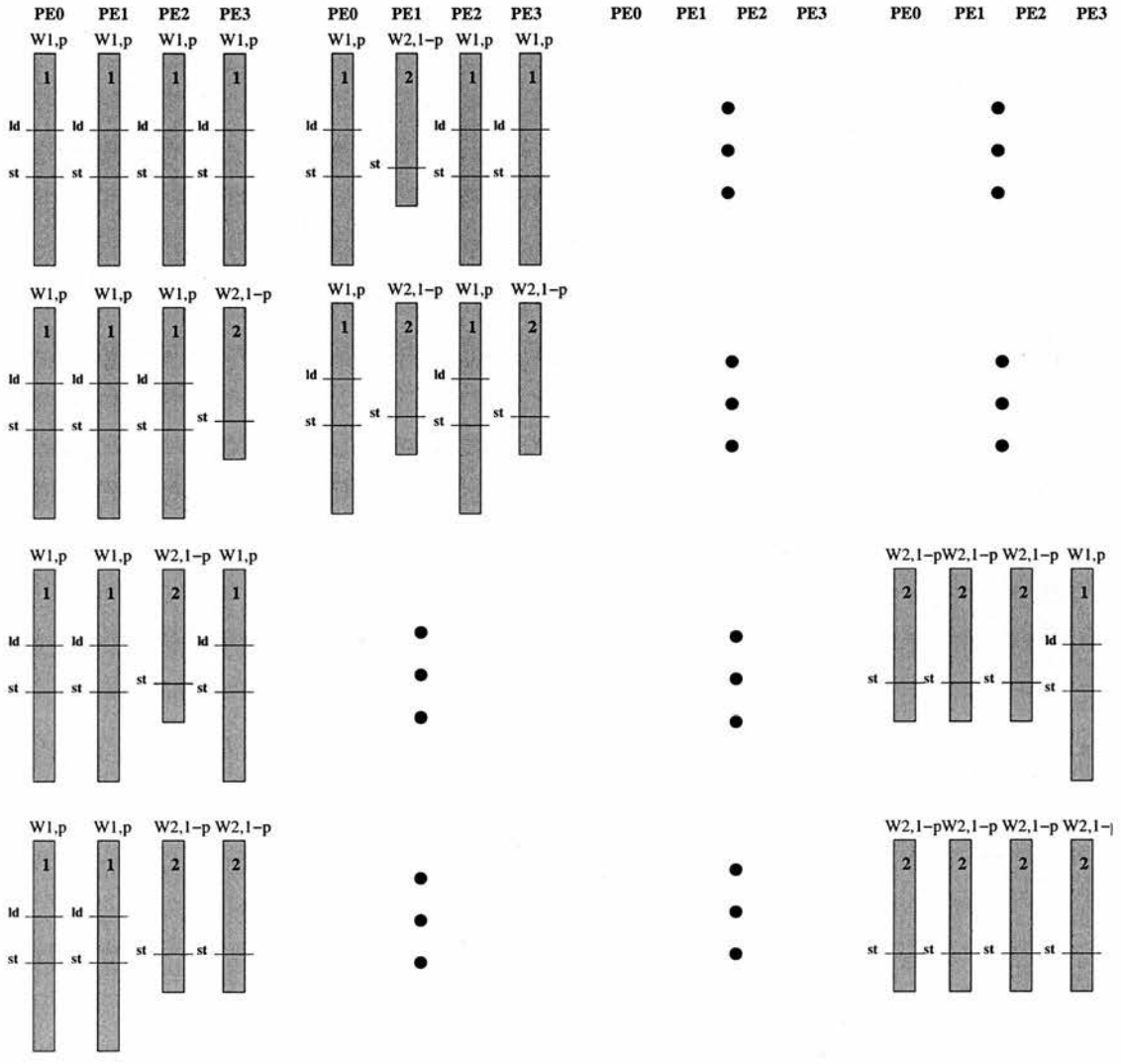

Figure 5.2: There are $2^4 = 16$ possible permutations of two thread sizes on four processors. Each of these permutations is called a *tuple*.

speculative execution, which is what one would expect from a sequential execution of the code section. To compute the overall estimated sequential execution time and the overall estimated parallel time, an average tuple which represents all the base tuples is computed using the weighted mean, *i.e.* the sum of all the base tuples' time of execution T_{tuple} with each weighted by its tuple probability p_{tuple}. To compute the overall estimated sequential execution time, only thread tuples formed by the subset $B \in A$ of base thread sizes are considered. Thus, the expected sequential execution time of the average tuple can be computed as:

$$Tseq_{est} = \sum_{tuple_i \in B^P} Tseq_{tuple_i} \cdot p_{tuple_i} \tag{5.3}$$

A naive computation of Equation 5.3 would involve enumerating all possible base tuples and would, thus, have a computational complexity of $O(|B|^P)$, where $|B|$ is the cardinality of set B, *i.e.* the number of base threads. However, it can be easily shown that since the assignment of thread sizes to processors in a thread tuple can be seen as independent discrete random variables from the same distribution, Equation 5.3 is equivalent to:

$$Tseq_{est} = P \cdot \sum_{W_i \in A} W_i \cdot p_i \tag{5.4}$$

which can be computed in $O(|B|)$.

To compute the estimated parallel execution time of the average tuple, it can be noted that the probability of the parallel execution time of a particular tuple i being equal to a particular thread size W_j is given by (recall that the thread sizes are sorted in increasing order):

$$p(Tpar_{tuple_i} = W_j) = \begin{cases} (\sum_{k=1}^{j} p_k)^P - (\sum_{k=1}^{j-1} p_k)^P, & (if\ 2 \leq j \leq N) \\ (p_1)^P, & (if\ j = 1) \end{cases} \tag{5.5}$$

Note that to compute the estimated average parallel execution time all N thread sizes are considered, including those that arise due to speculative execution overheads. Then, using Equations 5.2 and 5.5 it is possible to compute the expected parallel execution time of a tuple as:

$$Tpar_{est} = \sum_{j=1}^{N} p(Tpar_{tuple_i} = W_j) \cdot W_j \tag{5.6}$$

Equation 5.6 has a computational complexity of $O(N)$. Finally, using Equations 5.4 and 5.6 the estimated speedup can be computed as:

$$S_{est} = \frac{Tseq_{est}}{Tpar_{est}} \tag{5.7}$$

Section 5.4 shows an detailed example of how to use the above formulas. Finally, note that when the number of iterations in the loop is known to be less than the number of processors in the system, the value of P in the formulas above must be changed from the number of processors to the number of iterations.

5.3 Implementation Issues

5.3.1 Program Representation: The *Collapsed Control Flow Graph (CCFG)*

This section discusses how to compute the possible thread sizes that can be generated through the different execution paths, the *base thread sizes*. These thread sizes are intrinsic to the code structure and do not include speculative execution overheads.

To compute the base thread sizes, a variation of the control flow graph (CFG) was developed, called a *collapsed control flow graph (CCFG)*. To build the CCFG, each basic block node in the CFG is annotated with the estimated execution time of the instructions in this basic block. Also, when building the CCFG for a particular loop that is being considered for speculative execution, all its inner loops and procedures are collapsed into a subgraph with an estimation of the execution time of the control paths in these inner loops or procedures. In this way the CCFG of properly structured programs should not have any backward edges and is then a directed acyclic graph.

As an example, Figure 5.3a shows the skeleton of a loop in C-like syntax and Figure 5.3b shows the corresponding CCFG. Note how in the CCFG a loop is represented by an acyclic subgraph where the weights of the arcs are multiplied by the iteration count, M.

For the base thread sizes to be accurate, it is important that the CCFG be built from an intermediate representation whose basic block code is very close to the final

machine code to be produced by the code generator. On the other hand, some speculative parallelisation analyses and transformations are likely to be performed using some high-level representation. These conflicting requirements are addressed by using an intermediate representation of SUIF1 [18] that still retains much of the high-level information from the original source code while containing basic block code that is very similar to the final code. In a more integrated environment close collaboration between the code generator and the speculative paralleliser could lead to better accuracy.

With the CCFG the compiler can easily generate all the possible execution paths from the start node to the end node along with their respective estimated execution times and execution probabilities. Execution probabilities can be generated by simply assigning equal probability to each direction of a conditional statement or through some more elaborate static or dynamic mechanism for estimating the probabilities of the different directions [3]. The results indicate that the simple equal probabilities heuristic leads to reasonable results.

[3]Note that the problem of estimating the probability of conditional statements is usually more complex than the problem of estimating the probability of branches in general. This is because loop-controlling branches are usually more predictable than non-loop branches. This problem is also more complex than the simpler problem of branch prediction, which only involves estimating the most-likely direction of a branch.

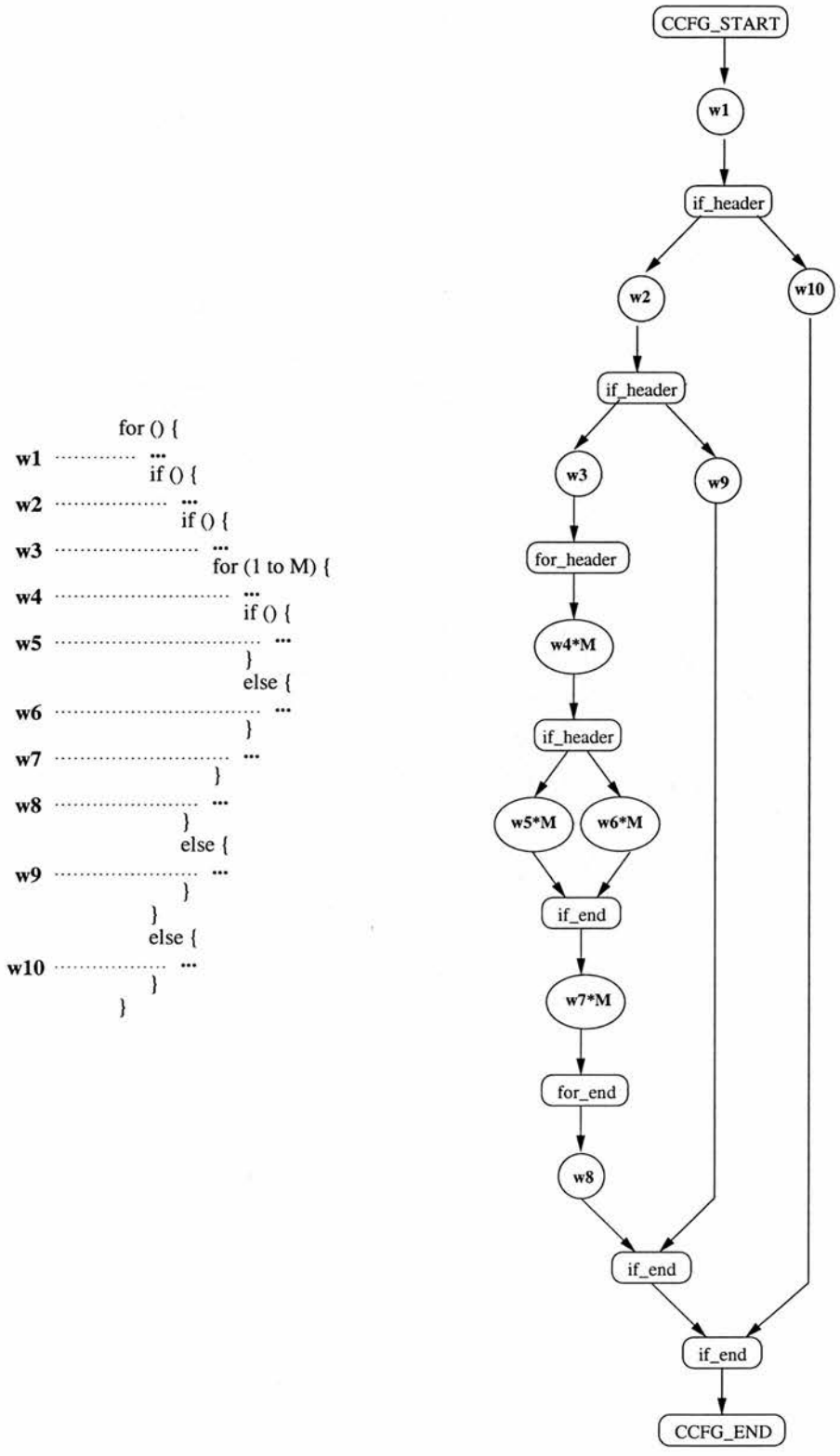

Figure 5.3: Example loop being considered for speculative parallel execution (a); and its corresponding collapsed control flow graph (CCFG) (b). The labels inside the basic blocks correspond to their estimated execution times. In this example M is the iteration count of the inner loop.

5.3.2 Base Thread Table

All the necessary information regarding the base thread sizes are stored in a table data structure which is defined in this thesis as *base thread table*. This table has three columns: base thread sizes (W_i), sequential probabilities (p_i) and parallel probabilities ($p(Tpar_{tuple_i} = W_j)$). The table is sorted in ascending order of base thread sizes. An example of the base thread table for the example shown in Figure 5.1 is presented in Table 5.1

W_i	p_i	$p(Tpar_{tuple_i} = W_j)$
$W_1(W2)$	$1-p$	$(1-p)^P$
$W_2(W1)$	p	$1-(1-p)^P$

Table 5.1: Base thread table for the example in Figure 5.1

The sizes and sequential probabilities are directly computed from the CCFG traversal algorithm, which is described in detail in Section 5.3.3. The parallel probabilities are computed based on the equations described in Section 5.2.

With the *squash and restart* overhead, Table 5.1 becomes Table 5.2, which is called the *extended thread table*. The new thread W_3, is an overhead thread, derived from the potential data dependence violation between the load operation to X[A[i]] in W_1 and the store operation to X[B[i]] in W_1. The details about the *extended thread table* and how the the overhead thread sizes are computed are discussed in detail in Part III. Since this part focuses on the evaluation of load imbalance overhead, only base thread table is needed to estimate it.

W_i	p_i	$p(Tpar_{tuple_i} = W_j)$
$W_1(W2)$	$1-p$	$(1-p)^P$
$W_2(W1)$	p	$(1-(1-p)^P)\cdot(1-p_{dep})$
$W_3(W1+w1+w2+w3)$	p	$(1-(1-p)^P)\cdot p_{dep}$

Table 5.2: Base thread table in Table 5.1 with a squashed thread.

5.3.3 The CCFG Traversal Algorithm

The base thread table is generated from the CCFG using a linear graph traversal algorithm. This algorithm is presented in Figure 5.4.

During the CCFG traversal, upon reaching an *if_header* node (Figure 5.3b), the graph search is split into two search paths. One along the `then` branch, and one along the `else` branch. When the search path from the `then` branch reaches the *if_end* node, it saves its search result, the current base thread table, into the *if_end* node and stops searching. When the search path from the `else` branch reaches the *if_end* node, it reads the base thread table previously saved by the `then` search path, and merges this table with its own base thread table and then continues searching to the next CCFG node. In the case when there is no `else` statement in the program, there is still an empty arc connecting the *if_header* and *if_end* nodes, in place of the `else` branch. Figure 5.5 shows how the table is duplicated in *if_header* nodes and merged in *if_end* nodes during the CCFG traversal.

In the algorithm, *memops* represents the memory access operations. They are necessary for the computation of the overhead thread sizes. Details about the *memops* are discussed in Section 7.4.1

```
Initialise base thread table B with an empty entry
Initialise procedure call stack S=0
traverseCCFG( CCFGNode n, tableType B)
  switch n.nodeType():
    if_header:
      create duplicate B' from B
      updateTableEntriesProbabilities (B , p_then)
      updateTableEntriesProbabilities (B', p_else)
      traverseCCFG(n.then, B)
      traverseCCFG(n.else, B')
    if_end:
      B = mergeTable(B, B')
      traverseCCFG(n.next, B)
    basic_block:
      collect memops of n
      increment all entries of B with n.size()
      traverseCCFG(n.next, B)
    loop_header:
      traverseCCFG(n.next, B)
    loop_end: (M iterations (if M>1))
      create memops for iterations [2,M]
      increment all entries of B with W_iteration · (M − 1)
      traverseCCFG(n.next, B)
    proc_call:
      if proc_call not already in call stack S
        push proc_call into call stack S
        traverseCCFG(proc_call, B)
```

Figure 5.4: Algorithm for building the base thread table from the CCFG

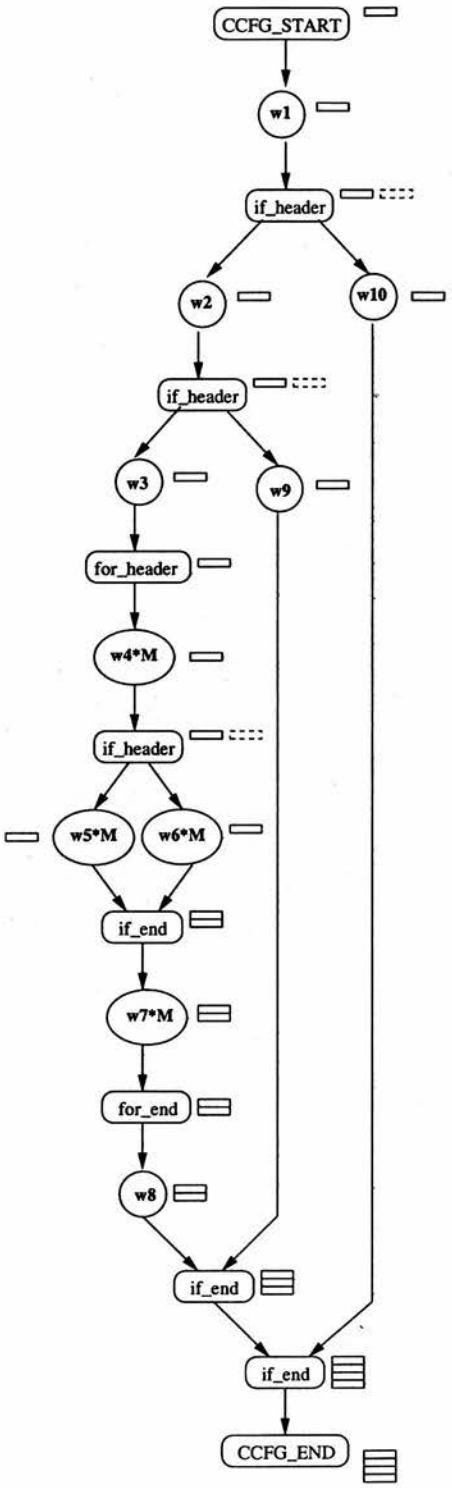

Figure 5.5: Duplicating and merging base thread table during the CCFG traversal

5.4 An Example Computation of S_{est}

As an example, consider the code section in Figure 5.3a and its CCFG in Figure 5.3b. The base thread sizes and their probabilities are shown in Table 5.3. Assume a system with four processors. Table 5.4 shows some of the $4^4 = 256$ possible thread tuples (4 thread sizes and 4 processors), along with their probabilities of occurrence and their resulting parallel and sequential execution times.

Thread size	*Probability of occurrence*	*Execution time*	*Example size*
W_1	50%	$w1 + w10$	100
W_2	25%	$w1 + w2 + w9$	150
W_3	12.5%	$w1 + w2 + w3 + w4 * M + w5 * M + w7 * M + w8$	220
W_4	12.5%	$w1 + w2 + w3 + w4 * M + w6 * M + w7 * M + w8$	370

Table 5.3: Base thread sizes and probabilities for the example in Figure 5.3. The thread sizes are sorted in increasing order (thus $w2 + w9 > w10$, $w3 + w4 * M + w5 * M + w7 * M + w8 > w9$, and $w6 > w5$). The values in the last column are arbitrary and only for the sake of the example.

Using Equation 5.5 the probabilities of the parallel execution time of a given thread tuple i can be computed:

$$
\begin{aligned}
p(Tpar_{tuple_i} = W_1) &= p_1^P = (0.5)^4 = 0.0625 \\
p(Tpar_{tuple_i} = W_2) &= (p_1 + p_2)^P - p_1^P = (0.5 + 0.25)^4 - (0.5)^4 = 0.2539 \\
p(Tpar_{tuple_i} = W_3) &= (p_1 + p_2 + p_3)^P - (p_1 + p_2)^P \\
&= (0.5 + 0.25 + 0.125)^4 - (0.5 + 0.25)^4 = 0.2698 \\
p(Tpar_{tuple_i} = W_4) &= (p_1 + p_2 + p_3 + p_4)^P - (p_1 + p_2 + p_3)^P \\
&= (0.5 + 0.25 + 0.125 + 0.125)^4 - (0.5 + 0.25 + 0.125)^4 \\
&= 0.4138
\end{aligned}
$$

Thread tuple				*Probability of*	*Parallel time*	*Sequential time*
PE_0	PE_1	PE_2	PE_3	*occurrence* (p_{tuple_i})	($Tpar_{tuple_i}$)	($Tseq_{tuple_i}$)
W_1	W_1	W_1	W_1	p_1^4	W_1	$4W_1$
W_1	W_1	W_1	W_2	$p_1^3 \cdot p_2$	W_2	$3W_1 + W_2$
W_1	W_1	W_1	W_3	$p_1^3 \cdot p_3$	W_3	$3W_1 + W_3$
W_1	W_1	W_1	W_4	$p_1^3 \cdot p_4$	W_4	$3W_1 + W_4$
W_1	W_1	W_2	W_1	$p_1^3 \cdot p_2$	W_2	$3W_1 + W_2$
W_1	W_1	W_2	W_2	$p_1^2 \cdot p_2^2$	W_2	$2W_1 + 2W_2$
W_1	W_1	W_2	W_3	$p_1^2 \cdot p_2 \cdot p_3$	W_3	$2W_1 + W_2 + W_3$
W_1	W_1	W_2	W_4	$p_1^2 \cdot p_2 \cdot p_4$	W_4	$2W_1 + W_2 + W_4$
W_1	W_1	W_3	W_1	$p_1^3 \cdot p_3$	W_3	$3W_1 + W_3$
W_1	W_1	W_3	W_2	$p_1^2 \cdot p_3 \cdot p_2$	W_3	$2W_1 + W_3 + W_2$
...	...	...	...	...	...	...
W_4	W_4	W_4	W_3	$p_4^3 \cdot p_3$	W_4	$3W_4 + W_3$
W_4	W_4	W_4	W_4	p_4^4	W_4	$4W_4$

Table 5.4: Some thread tuples for the example in Figure 5.3 running on four processors. For each thread tuple its probability of occurrence and its resulting parallel and sequential execution times are given (recall that thread sizes are sorted in increasing order).

With $p(Tpar_{tuple_i} = W_j)$, the base thread table(Section 7.4) for this example can be generated. It is shown in Table 5.5

W_i	p_i	$p(Tpar_{tuple_i} = W_j)$
$W_1(100)$	0.5	0.0625
$W_2(150)$	0.25	0.2539
$W_3(220)$	0.125	0.2629
$W_4(370)$	0.125	0.4138

Table 5.5: Base thread table for the example in Figure 5.3.

Then $Tseq_{est}$, $Tpar_{est}$, and S_{est} can be computed using Equations 5.4, 5.6, and 5.7:

$$\begin{aligned} Tseq_{est} &= 4 \cdot \sum_{i=1}^{4} W_i \cdot p_i \\ &= 4 \cdot (100 \cdot 0.5 + 150 \cdot 0.25 + 220 \cdot 0.125 + 370 \cdot 0.125) = 645 \end{aligned}$$

$$\begin{aligned} Tpar_{est} &= \sum_{j=1}^{4} p(Tpar_{tuple_i} = W_j) \cdot W_j \\ &= 0.0625 \cdot 100 + 0.2539 \cdot 150 + 0.2698 \cdot 220 + 0.4138 \cdot 370 = 257 \end{aligned}$$

$$S_{est} = \frac{Tseq_{est}}{Tpar_{est}} = \frac{645}{257} = 2.51$$

To check the accuracy of the model for this example, a sequence of 1 million threads was generated, with the sizes and probabilities shown in Table 5.3 and these were fed to the trace simulation environment, described in Section 6.1.3. The speedup obtained in the simulation was 2.98, different by only 19% from the speedup predicted with the model. The model predicts a smaller speedup mainly because of the limitation of Equation 5.2, where a processor that has completed the execution of a thread that is predecessor to the longest thread in the tuple is not allowed to proceed with a new thread. The simulation correctly handles this case.

5.5 Limitations of the CCFG

As described, the CCFG does not correctly represent the execution paths and execution times in the presence of inner loops. An inner loop with M iterations and with conditional structures leading to N possible control paths can generate up to N^M combinations of paths at run time. Instead of trying to represent these (prohibitively) many paths in the CCFG, a compromise is to represent only a total of N possible paths that are executed M times (Figure 5.3). This corresponds to the case where all iterations in the inner loop execute the same path in a given execution of the inner loop (different execution instances of the inner loop can follow different paths).

Another weakness of the CCFG is that it cannot easily handle recursive procedure calls. In the presence of recursion, a compromise is to represent in the CCFG only a limited number of the recursive calls. Thus, it is proposed to handle recursion as follows. When a recursive call is detected, a decision is made on whether to build the procedure's subgraph depending on the probability of occurrence of the path leading to the recursive call. If the probability is below a threshold then the procedure call is simply ignored and it is assigned a null workload. Note that recursion can be easily detected during the construction of the CCFG by keeping a stack of procedure names while building the CCFG.

One assumption made is that while traversing the CCFG, if the destination of a `goto` statement jumps to outside the loop's CCFG, it is considered as the termination point of the current execution path.

5.6 Optimisations for CCFG Traversal

As an optimisation of the CCFG traversal, thread sizes on paths leading directly to `exit` and `break` statements are ignored. This is reasonable since these paths lead to the termination of the speculative execution and, thus, should not be included in the thread size mix.

A potential problem of this compiler models is the memory usuage. When the speculative section has several possible execution paths and many potential overheads this model may result in too many thread sizes. A particularly difficult case is when the code consists of a series of conditional statements. In this case the number of base thread sizes will be exponential with respect to the number of conditional statements.

An optimisation technique is proposed to cut the size of the table dynamically as the the CCFG traversal algorithm progresses along the $CCFG$. The idea is that threads whose sizes differ by only a small percentage can be merged together. The resulting thread size could be as long as the average of the individual thread sizes and the resulting probability of occurrence would be the sum of the probabilities of the thread sizes.

In this optimisation, when two tables are merged together in the if_end node during the CCFG traversal, a threshold is applied on the size of the resulting table. If this table size exceeds a predefined number, then two adjacent table entries which have similar thread sizes will be merged into a single entry. This process is repeated until the table size shrinks to the threshold value. The algorithm of this *mergeTable* is shown in figure 5.6.

```
#define THRESHOLD 1000
mergeTable(tableType A, tableType B)
{
   merge A and B, then sort by thread size
   for each adjacent pair of entries, i and i-1:
      compute thread sizes relative difference: diff_i = |(W_i - W_{i-1}) / (W_i + W_{i-1})|
   sort table by diff_i, in increasing order
   while(table_size > THRESHOLD)
      merge_pair(entry_{table_size-1}, entry_{table_size})
}
merge_pair(tableEntry i, tableEntry j)
{
   W_i = (W_i + W_j) / 2
   p_i = p_i + p_j
   transfer memops of j to i
   delete tableEntry j
}
```

Figure 5.6: Algorithm for mergeTable

Chapter 6

Evaluation

6.1 Experimental Setup

6.1.1 Compilation Infrastructure

The algorithms and formulas described in Chapter 5 were implemented in the SUIF1 compiler infrastructure [18]. All the compiler analyses are done at the intermediate representation (IR) of SUIF. There are two different formats in SUIF1, namely, the high-SUIF format and the low-SUIF format. The program was initially compiled into high-SUIF format, and then selectively dismantled info low-SUIF format. The advantage of the high-SUIF format is that it retains high-level control structures such as `if`, `for loops`, and their loop iteration counts. However, there are complex instructions, such as memory copy, and modular which are not commonly supported. The low-SUIF format has no high-level control structure, but, the resulting IR is close to the machine assembly format. In the experiment, the IRs in each basic block node are split into low-SUIF format using the SUIF1 utility *porky*. The resulting IR is convenient because it retains all the relevant information from the source program while containing individual instructions very close to the machine code. A new pass was implemented in SUIF1 and was added at the end of the *pscc* chain, which is the SUIF parallelising compiler.

In the evaluation of this part, only the tuple model and the heuristics for estimating load imbalance are considered. Part III evaluates the extended tuple model with squash and restart overhead. Both evaluations include the dispatch and commit overhead assuming that the nominal commit time is a fixed time required to write back

all the entries in the speculative buffer with some hardware support [19] and that the dispatch overhead is also a fixed time as suggested in previous work [19, 42]. A more detailed explanation of how dispatch and commit overheads are modelled is presented in Section 7.3.5. The iteration count for the inner loop was assumed to be one for the loops with statically unknown upper bounds in this part of the work. In the evaluation of the extend tuple model of part III, based on the research result of [50], five iterations are assumed for loops with statically unkown upper bounds.

6.1.2 Applications

A subset of the SPEC2000 benchmarks, comprising of 4 floating point and 5 integer applications [1] is used to evaluate the scheme. These applications are representative of the workloads typical of workstations, which are the typical market for current and future CMP's. To keep the execution time in the simulation environment (Section 6.1.3) manageable, the reduced input sets for SPEC benchmarks [22] is used and the experiment only simulated 500 million instructions after discarding the initial 100 million instructions. For each application only loops that are not parallelised by SUIF1 are considered, as these are likely to be more profitably executed in parallel without speculation.

To focus on the tuple model and the heuristics for estimating load imbalance and squash and restart overheads, loops that overflow the speculative buffer are not considered at run time. Additionally, to isolate the tuple model and load imbalance overhead from the heuristics for squash and restart overhead the loops are separated into two groups: those that are identified, by manual inspection, as not having dependences and those having dependences. Note that loops in the latter group may or may not suffer data dependence violations at run time due to the actual timing of speculative loads and stores and the scheduling of threads. The loops in the first group are used to evaluate the tuple model and the load imbalance overhead in this part. They correspond to those loops used in [13]. To estimate the accuracy of the model we consider for speculative parallelization all loops except those that are parallelizable by SUIF1 alone (we call these Doall loops). However, when reporting speedups we also do not consider for speculative parallelization loops that are inside Doall loops or are inside loops chosen

[1] The benchmarks not included are those incompatible with the SUIF1 front end pass, *snoot*, which translates the pre-processed C program to the SUIF intermediate format.

for speculative parallelization (this is because our architecture does not support nested parallelism). Table 6.1 lists the applications we use along with the number of loops we consider for speculative execution (the numbers in parenthesis are those discounting inner loops), the fraction of the sequential execution time that is taken by these loops, and the fraction of the sequential execution time that is taken by loops parallelized by SUIF1.

Application	*Benchmark type*	*Number of loops*		*% of seq. time*	
		Without dep.	*With dep.*	*spec*	*doall*
177.mesa	floating point	7	2	93	0
179.art	floating point	10 (5)	8 (6)	99	≈ 0
183.equake	floating point	28 (18)	24 (16)	74	26
188.ammp	floating point	4	1	50	0
175.vpr	integer	4	5 (4)	96	≈ 0
181.mcf	integer	4	3	40	0
186.crafty	integer	16	16	23	0
255.vortex	integer	4	0	< 1	0
256.bzip2	integer	24 (17)	37 (29)	91	5

Table 6.1: Characteristics of the applications studied.

6.1.3 Architecture Simulated

Several architectures for speculative execution in CMP-like systems have been proposed (e.g., [2, 19, 23, 27, 32, 41, 42, 46]). This thesis assumes a CMP in the lines of the Stanford Hydra CMP [19, 20]. This system consists of four single-issue processors, each with a private L1 data cache attached to a shared on-chip L2 cache with separate read and write buses. Each processor has also a private fully-associative speculative buffer. The bus protocol supports both cache coherence and speculative parallelisation. Table 6.2 shows the configuration parameters of the system modelled, which are similar to those listed in a recent publication on the Hydra CMP [36].

Processor Param.	*Value*
Number of processors	4
Issue width	1
Memory Param.	*Value*
L1, L2 size	16KB, 2MB
L1, L2 assoc.	4-way, 4-way
L1, L2 line size	32B, 64B
L1, L2 latency	1, 5 cycles
Spec. buffer size	2KB
Spec. buffer assoc.	full
Spec. buffer latency	1 cycle

Table 6.2: Parameters of the speculative CMP modelled.

6.1.4 Simulation Design: Initial Investigation

At the time of developing the simulation infrastructure we performed a thorough investigation of all then available architectural simulators. As somewhat expected, no simulator that incorporated speculative multithreading was publicly available at the time. Moreover, most simulators either did not offer support for multiprocessors or did not offer a detailed microarchitectural execution model. Developing a detailed simulator with speculative multithreading and microarchitectural model was deemed not practical within the timeframe of this thesis, especially since a large development effort would also need to be made into the compilation infrastructure. Thus, some compromises had to be made. The decision was then to use the Virtutech's Simics [26] simulator without a microarchitectural model and to resort to trace-driven simulation. Despite trace-driven simulation's well known shortcomings for simulation of parallel programs in multiprocessors, these are likely to produce timing differences that are significantly overshadowed by the differences caused by the main overheads of speculative multithreading execution. Likewise, any timing difference from the lack of a detailed microarchitectural model is likely to produce only second-order effects. Overall, considering the amount of time for the thesis work and the fact that the focus of the thesis work is on the compiler, and not on architectural and microarchitectural features,

I believe that this was a reasonably sound choice.

6.1.5 Simulation Mechanism

The simulator was designed and implemented using the trace-driven mechnism. The simulation process is composed of two stages, the trace generation stage, and the trace processing and analyzing stage. During the trace generation stage, the program was executed on the Virtutech's Simics [26] which is a cycle accurate full system simulator, which simulates a desktop computer running Linux Enterprise. At the time of implementing the simulator, due the lack of a complete support for multiprocessor simulation, the decision was made to simplify the trace generation by running the benchmarks on a single processor, which was configured according the Hydra CMP in terms of cache sizes, associatities, and latencies, etc. Then a separate, custom-made, trace analyzer was develop to mimic the speculative multhreading execution.

The benchmark programs are firstly compiled with a host compiler and then transfered to the target simulator for execution. Simics is a full system simulator and it simulates machine behavior of both the user and operating system code. To account for only the application execution, the tracing facilities are dynamically turned off during supervisor mode. During the program execution, traces are written to a trace file. Before compiling the program, Simics' magic instruction sequences are inserted at the beginning and the end of each loop construct in the source programs. These magic instructions trigger the Simics system haps, which turn on and off the tracing facilites to print out: the cycle number and addresses of the memory accesses, the cycle numbers of the beginning and the end of each iteration of the loop, and an unique ID of the loop identified by its line number and the file name. This magic instructions, however, have no visible effect on the executed program or the simulator statistics.

During the second stage of the simulation (after the trace file has been generated), the trace file is fed into a trace analyzer which computes the value of the speedup over sequential execution on a speculative multithreading CMP enviromment for each loop. In order to compute the speedup of a loop, the trace analyzer computes the total sequential execution time and the total parallel execution time of this loop. The sequential time of the loop can be easily computed by the cycle differences between the last cycle of the last iteration of the loop and the first cycle of the first iteration of the loop.

The parallel execution times are generated by computing the parallel execution time of the loop, when the loop iterations are running as parallel threads on a speculative multithreading CMP. Thus, consecutive iterations of a loop are assigned to virtual processors as if these iterations were running in parallel. The trace analyzer reads in consective iterations one at a time, and computes the speculative execution time for each processor based on the run time speculative multithreading behavior. For instance, squash behavior is based on comparing the address and timestamps of the memory operations of these iterations. More specifically, memory operations of each iteration are compared against the memory operations to the same address from other concurrent iterations to detect any cross-iteration data dependence violations. If violations are detected, the offending thread is squashed together with all its concurrent successors threads. This squashed behavior will be reflected as the additional time on that processor.

The thread dispatch and commit overhead is a variable overhead which is usually determined by the amount of speculative data that needs to be committed to the memory. The previous research in the Hydra group showed that this is about 12 clock cycles [7] on average. For the simplicity of implementation, this overhead is assumed to be a fixed 12 clock cycles in the simulator implementation. Thus, after computing the processor time of a thread, each thread's execution time is incremented with 12 clock cycles to account for the commit overhead.

After analyzing a concurrent group of iterations, the trace analyzer updates the execution time of each processor and proceeds to read in and process the next batch of iterations. After all the iterations of the loop have been computed, the parallel execution time on each processor can be computed as the total execution time of all the iterations that have been assigned onto this processor. The total parallel time of the loop is equal to the final execution time of the processor which executed the last iteration of the loop.

6.1.6 Limitations of the Simulation Environment and its Inaccuracies

Because the trace file was generated from a single processor execution of the program, some behaviors that can appear with actual concurrent execution are missed out. These include bus contention and delays in the shared L2 cache due to conflict miss. The bus

contention latency happens when the load or store operations from different processors happen at the same time, and they compete for the common read or write bus. In the worst case scenario, all the processors are competing for the write bus at the same time, thus a store from a processor might have to wait for the completion of all the other processors' requests before it can start. On the other hand, the best case is that no processor is using the write bus, and the store operation can proceed without delay.

Another inaccuracy of the simulation is due to the conflict misses on the shared L2 cache. This event happens when two load operations from different processors conflict on the same L2 cache line. Thus the value has to be reloaded into the L2 cache, when the second load operation proceeds. This behavior is difficult to integrate int the simulation framework, so for simplicity, this effect is not simulated.

6.2 Speculative Parallelisation Performance

To assess how often speculative parallelisation leads to speedups or slowdowns with respect to sequential execution, Figure 6.1 shows, for each application, a histogram of the actual speedups obtained with the simulations for 4 processors. This figure shows that speculative parallelisation leads to a broad range of speedups and slowdowns: on average 54% of the loops lead to slowdowns while 46% lead to speedups. These results evidence the importance of being able to selectively apply speculative parallelization.

6.3 Model Accuracy

The output of the compiler prediction model is an estimate of the actual value of the speedup or slowdown. The primary use of this prediction is to switch off speculative parallelisation of loops that are expected to lead to slowdowns and speculatively parallelise loops that are expected to lead to speedups. Figure 6.2 shows, for each application, a breakdown of the fraction of loops that are correctly predicted as leading to speedups (*Predict speedup/Actual speedup*), that are correctly predicted as leading to slowdowns (*Predict slowdown/Actual slowdown*), that are incorrectly predicted as leading to speedups (*Predict speedup/Actual slowdown*), and that are incorrectly predicted as leading to slowdowns (*Predict slowdown/Actual speedup*).

This figure shows that the model is accurate enough to correctly identify the speedup

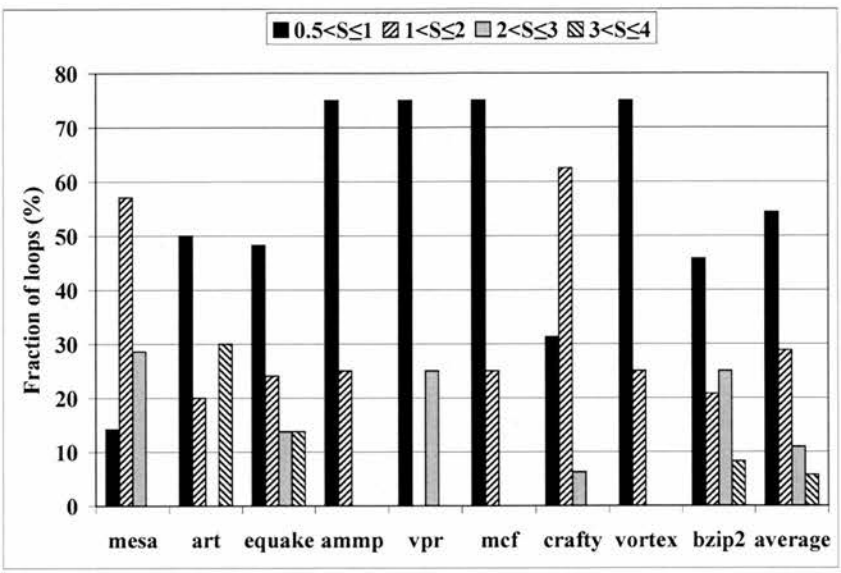

Figure 6.1: Histogram of speedups for the speculative loops only. Note that a speedup (S) less than one is a slowdown.

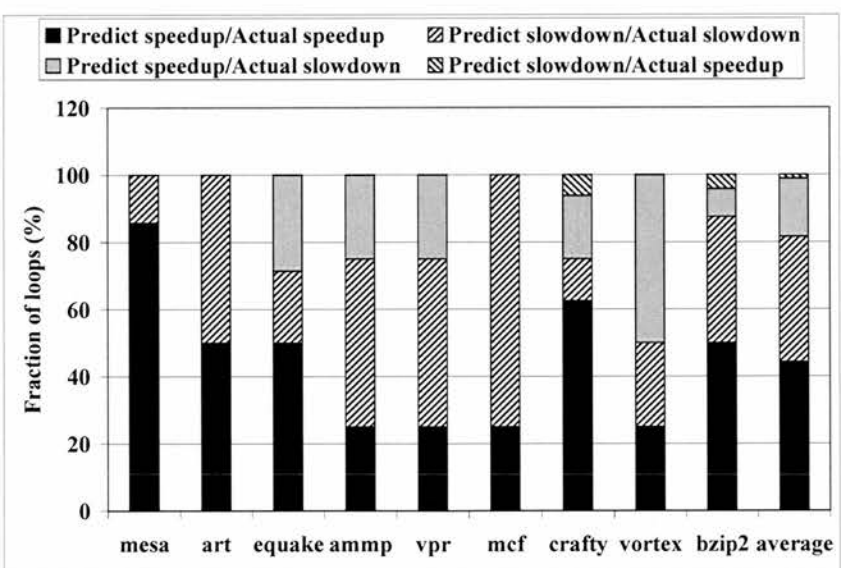

Figure 6.2: Outcome of predictions with respect to actual observed speedup or slowdown.

or slowdown outcome of the speculative execution in 82% of the cases, on average. Also, it seems that when the model is incorrect the tendency is to overestimate the performance gains. This means that the model rarely misses a valid opportunity to speculatively parallelize loops, but that it does not prevent all of the performance degradation from slowdown cases.

In some cases it may be useful to have an accurate prediction of the actual value of the speedup or slowdown. Figure 6.3 shows the cumulative error distribution for each application plotted at 10% boundaries. The errors are computed as the absolute difference between the estimated speedups and the actual speedups obtained with the simulation, divided by the actual speedup. Thus, in this figure a point (x, y) in the curve means that $y\%$ of the loops have errors less than $x\%$.

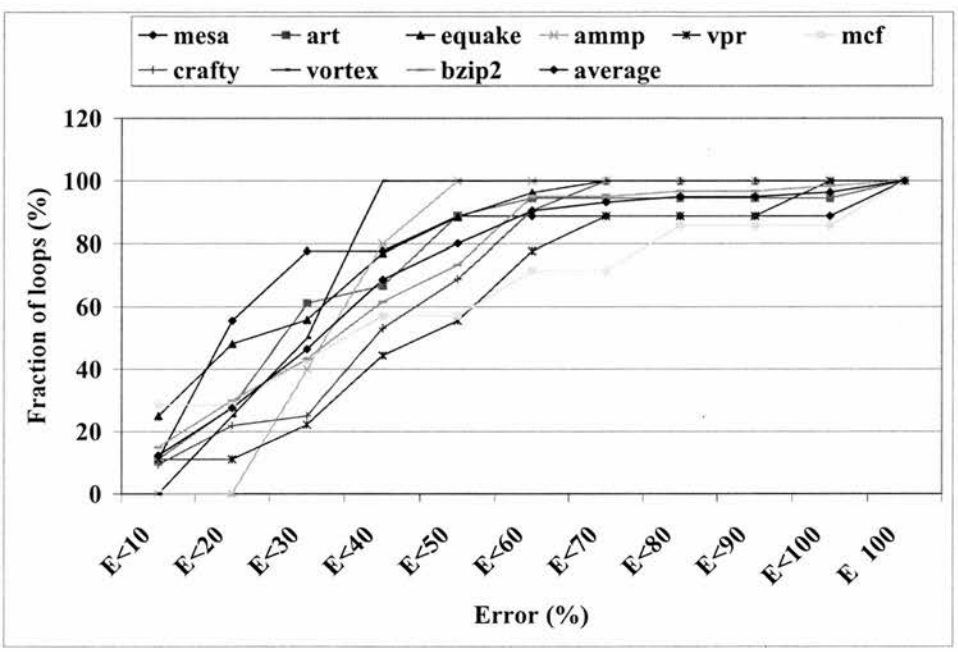

Figure 6.3: Cumulative error distributions.

This figure shows that the model is reasonably accurate and is able to predict the actual speedup or slowdown within 20% for 44% of the loops on average and within 50% for 84% of the loops on average.

A closer look at the results shows that the largest errors appear on loops that lead to

slowdowns, in which case the model estimates a larger slowdown than what is actually observed. These cases tend to relate to relatively small loops and the reason for this overestimation of the slowdowns can be attributed to differences between the thread sizes estimated from the SUIF IR and the actual thread sizes. Table 6.3 lists the sources of errors for the top 10% loops with the largest errors (10 loops in total), along with the number of times these sources created such large errors.

Description of source	*Number of instances*	*Range of errors (%)*
Incorrect thread size estimation from SUIF IR	4	54 to 116
Unknown iteration count of spec loop (and $< P$)	3	54 to 61
Unknown iteration count of inner loop	2	98 to 161
Biased conditional	1	136

Table 6.3: Observed sources of prediction errors for the top 10% of loops with the largest errors.

This table shows that the reasons for the very large prediction errors are either the incorrect workload computed from the SUIF intermediate representation or the lack of information at compile time.

6.4 Performance Improvements

Finally, this section estimates the actual performance impact of using the model to speculatively parallelise only loops that are predicted to have speedups. Figure 6.4 shows the overall speedups obtained for the execution of all the loops under consideration. In this plot a speedup of one means an execution time equal to the sequential execution time of the loops. This figure shows the result of speculatively parallelising loops following the predictions of the framework (*Selective*) and following a policy to speculatively parallelise all the loops (*Naive*). The figure also shows the result of following a simple selection policy based on a minimum predicted thread size of 50

instructions (*Threshold*). Such policy has been used in previous work (e.g. [49]) as a heuristic to amortise any potential overheads from speculative multithreaded execution. Finally, the figure also shows the result of selecting the loops based on a perfect knowledge of the expected speedups and slowdowns (*Oracle*).

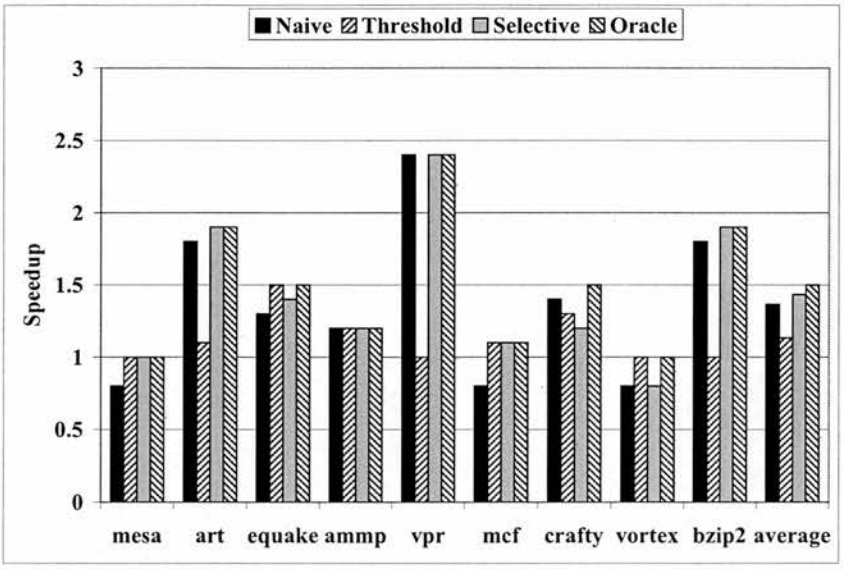

Figure 6.4: Overall performance gains for the speculative loops only.

This figure shows that the selections based on the results of the model consistently outperform *Naive* except for *186.crafty*, and *Threshold* except for *183.equake*, *186.crafty*, and *255.vortex*. The performance gains of the model over these schemes are 5% and 26% on average and as high as 38% and 140%, respectively. Moreover, the selection policy driven by the model achieved performance very close to *Oracle* in most cases, and within 5% on average. This was expected since the model incorrectly predicts slowdowns in less than 2% of the cases on average (Figure 6.2). Finally, it can be noted that the selection policy can significantly curb the performance degradations of speculative parallelization that occur for some applications with *Naive*. Again, this was expected since the model incorrectly predicts speedups in only 17% of the cases on average (Figure 6.2).

Part III

Extended Tuple Model and Other Overheads

Chapter 7

The Extended Framework

7.1 Reviewing the Speculation Overheads

Part II introduced the base thread tuple model, which handles well load imbalance and is also flexible enough to enable the integration of all the major speculation overheads. Chapter 2 has introduced the major speculation overheads. They are summarised in Table 7.1.

Overhead	*Description*
Load Imbalance	The time idle waiting for commit
Squash and Restart	Redundant re-execution after a violation is detected
Speculative Buffer Overflow	The time the thread is stalled due to the overflow of its speculative buffer
Dispatch and Commit	The time to dispatch the threads, and to commit the threads
Inter-thread Communication	The time for the data to be generated and communicated to the consumer thread

Table 7.1: List of all the major speculation overheads

This chapter introduces the extended framework, which is based on the work in Part II, but it much better suited and is more accurate to model other speculative parallelisation overheads. This chapter also describes in detail how each of these overheads

is integrated into the thread tuple model.

7.2 The Extended Thread Tuple Model

The tuple model proposed in [13] and Part II starts with all possible thread sizes and computes the probabilities associated with all possible tuples. In doing so, that model assumes that the probabilities that a certain thread size appears in a processor slot in a tuple does not depend on the processor slot. A more accurate model should consider how the probabilities of occurrences of thread sizes vary with the processor slot. For instance, a thread running in processor zero (PE_0), which is always assumed to be the non-speculative thread in a tuple, cannot have a size that corresponds to a thread that overflows the speculative write buffer. Similarly, the probability that a thread size that corresponds to a thread that has been squashed appears in a certain processor slot in a tuple actually depends on the processor slot: the probability that this squashed size appears in a more speculative processor slot is greater than the probability that it appears in a less speculative processor, since in the first case there are more chances that the producer thread will appear in some of its predecessor processor slots. Here the original tuple model is extended to accommodate these variations in probability values.

In the tuple model, the possible thread sizes are divided in two groups: the *base thread sizes*, which are those that correspond to the overhead-free execution of all possible execution paths, and the *overhead thread sizes*, which are those that are generated by adding some overhead to a base thread size. Thus, by definition, every overhead thread size has a corresponding base thread size from which it originates.

Chapter 5 showed how to compute the base thread table using a *collapsed control flow graph (CCFG)*. The major difference between the extended tuple model and the base tuple model is that the base tuple model only has one probability for each base and overhead thread size, whereas the extended tuple model properly takes into account the variations of the probabilities across processor slots. This chapter discusses how the base tuple model of Section 5.2 is extended to take this factor into consideration. As the extended tuple model is still based on the base tuple model, some of the concepts defined in Chapter 5 are repeated here for convenience.

7.2.1 Base Tuple Model Review

Like in the base tuple model, assume that the possible control flow paths originate thread sizes in an ordered set B, with M (or $|B|$) entries. Further assume that B joined with the overhead thread sizes form a new ordered set E which has N (or $|E|$) possible thread sizes $W_1, ..., W_N$. Thus, $E = \{W_1, W_2, ..., W_N\}$ and $W_1 \leq W_2 \leq ... \leq W_N$.

In the extended tuple model, the probability of occurrence of a particular thread tuple i, which is referred to by p_{tuple_i}, is the product of the probabilities of occurrences of each thread size in *a particular processor slot* in the tuple. These p_{tuple_i} are then used to compute the overall estimated sequential and parallel execution times as:

$$Tseq_{est} = \sum_{tuple_i \in B^P} Tseq_{tuple_i} \cdot p_{tuple_i} \tag{7.1}$$

$$Tpar_{est} = \sum_{tuple_i \in E^P} Tpar_{tuple_i} \cdot p_{tuple_i} \tag{7.2}$$

where $Tseq_{tuple_i}$ and $Tpar_{tuple_i}$ are the sequential and parallel execution times for tuple i and $B \subset E$ is the ordered set of the base thread sizes, as described above. $Tseq_{tuple_i}$ and $Tpar_{tuple_i}$ can be computed with Equation 5.1 and Equation 5.2, which are repeated here for convenience.

$$Tseq_{tuple_i} = \sum_{W_j \in tuple_i} W_j \tag{7.3}$$

$$Tpar_{tuple_i} = \max_{W_j \in tuple_i} W_j \tag{7.4}$$

As in Section 5.2, Equation 7.1 can be simplified to:

$$Tseq_{est} = P \cdot \sum_{W_i \in B} W_i \cdot p_i \tag{7.5}$$

which can be computed in $O(M)$, as in Section 5.2.

Similarly, to simplify the computation of Equation 7.2 it is converted using Equation 7.4 into:

$$Tpar_{est} = \sum_{W_j \in E} W_j \cdot p(Tpar_{tuple_i} = W_j) \tag{7.6}$$

where the second term is the probability that the parallel time of a tuple is equal to a given thread size. In the base tuple model of Section 5.2 this term was computed

through Equation 5.5. In the extended tuple model, to derive this term, a new term $p_{i,j}$ is defined, as the probability that thread size $W_i \in E$ appears in processor j in a given tuple (the processor slots are numbered from 0 to $P-1$). The computation of these $p_{i,j}$ terms is explained next.

7.2.2 Probabilities of Occurrence on Processor Slots

This section explains how to compute the key element of the extended tuple model, namely the probabilities that thread sizes appear in each processor slot in the tuple (i.e., $p_{i,j}$). A detailed description of how to generate the overhead threads and its rationale are given in Section 7.3.

The computation of all $p_{i,j}$ is divided according to the type of the thread W_i.

Case 1: W_i is a base thread size and it does not generate any new thread sizes with overheads.

In this case the probabilities are given by:

$$p_{i,j} = p_i \tag{7.7}$$

where p_i, as mentioned earlier, is simply the probability of execution of the control path that leads to the base thread size W_i.

Case 2: W_i is a thread size related to squash overheads (ie., either an overhead thread size generated from squash or the corresponding base thread size).

In this case W_{prod} is defined as the base thread size that produces the value that causes the violation (and, thus, p_{prod} is its probability), and W_{base} is defined as the base thread size that consumes the value and may suffer the violation. Then, the probabilities of both the squashed and the original, non-squashed, thread sizes are given by:

$$p_{i,j} = \begin{cases} (1-p_{prod})^j \cdot p_{base} + (1-(1-p_{prod})^j) \cdot p_{base} \cdot (1-p_{dep}) \\ \quad \text{, if } W_i \text{ is a base size} \\ (1-(1-p_{prod})^j) \cdot p_{base} \cdot p_{dep}, \quad \text{if } W_i \text{ is a squashed size} \end{cases} \tag{7.8}$$

where p_{base} is simply the probability of execution of the control path that leads to base thread size W_{base} and p_{dep} is the probability that the dependence occurs in an execution instance of the pair W_{prod} and W_{base}. Note that for a base thread size W_i and its squashed counterpart W_k there is $p_{i,j} + p_{k,j} = p_i$ for every processor j. Note also that $p_{i,0} = 0$ for a squashed thread size W_i.

Case 3: W_i is a thread size related to overflow overheads (ie., either an overhead thread size generated from overflow or the corresponding base thread size).

In this case, define: W_{base} as the base thread size that may overflow the buffer and become stalled; $firstlonger$ as the rank in B of the first thread size whose execution time is greater than the time of occurrence of the store that causes the overflow and the stall (a thread size W_i that overflows the speculative write buffer at time t can only stall due to predecessors of size greater than t); and, for a stalled thread size, $waitfor$ as the rank in B of the predecessor thread for which the stalled thread has to wait. Note that for a given overhead thread size W_i, $waitfor$, $base$, and $firstlonger$ are unambiguously defined. The probabilities of the overflowed and the original thread sizes are given by:

$$p_{i,j} = \begin{cases} (1-p_{ovflow}) \cdot p_{base} + p_{base} \cdot p_{ovflow} \cdot (1-\sum_{k=firstlonger}^{|B|} p_k)^j \\ \quad \text{, if } W_i \text{ is a base size} \\ ((\sum_{k=1}^{waitfor} p_k)^j - (\sum_{k=1}^{waitfor-1} p_k)^j) \cdot p_{ovflow} \cdot p_{base} \\ \quad \text{, if } W_i \text{ is an overflowed size} \end{cases} \tag{7.9}$$

where p_i is simply the probability of execution of the control path that leads to base thread size W_i and p_{ovflow} is the probability that the overflow occurs in an execution

instance of W_{base}. Note that for a base thread size W_i and its overflowed counterparts $W_{k_1}, ..., W_{k_n}$ (each for a different longest predecessor *firstlonger*), $p_{i,j} + p_{k_1,j} + ... + p_{k_n,j} = p_i$. Note also that $p_{i,0} = 0$ for an overflowed thread size W_i.

Case 4: W_i is a thread size related to inter-thread communication overhead (ie., either an overhead thread size generated from waiting for communication or the corresponding base thread size).

In this case define W_{prod} as the base thread size that produces the value that is involved in the communication (and, thus, p_{prod} is its probability), and W_{base} as the base thread size that consumes the value and may have to wait for the communication. When the inter-thread communication is set up dynamically based on data addresses (Section 7.3.4 and [10, 30, 44]) the communication, and, thus, the stall, will only occur when the dependence does occur. Then, the probabilities of both the stalled and the original, non-stalled, thread sizes are given by:

$$p_{i,j} = \begin{cases} (1-p_{prod})^j \cdot p_{base} + (1-(1-p_{prod})^j) \cdot p_{base} \cdot (1-p_{dep}) \\ \quad \text{, if } W_i \text{ is a base size} \\ (1-(1-p_{prod})^j) \cdot p_{base} \cdot p_{dep}, \quad \text{if } W_i \text{ is a squashed size} \end{cases} \tag{7.10}$$

which is equivalent to Equation 7.8 and where p_{base} is simply the probability of execution of the control path that leads to base thread size W_{base} and p_{dep} is the probability that the dependence occurs in an execution instance of the pair W_{prod} and W_{base}.

When the inter-thread communication is set up statically based on the static memory references (Section 7.3.4 and [23, 41]) the communication will always occur and it is assumed that a suitable producer will always be present to avoid deadlocks. In this case the probabilities of both the stalled and the original, non-stalled, thread sizes are given by:

$$p_{i,j} = \begin{cases} p_{base} & \text{, if } W_i \text{ is a base size and } j = 0 \\ 0 & \text{, if } W_i \text{ is a base size and } j \neq 0 \\ 0 & \text{, if } W_i \text{ is a stalled size and } j = 0 \\ p_{base} & \text{, if } W_i \text{ is a stalled size and } j \neq 0 \end{cases} \tag{7.11}$$

where p_i is simply the probability of execution of the control path that leads to base thread size W_i. Note that in Equations 7.10 and 7.11 for a base thread size W_i and its stalled counterpart W_k, $p_{i,j} + p_{k,j} = p_i$ for every processor j. Note also that $p_{i,0} = 0$ for a stalled thread size W_i in both equations.

7.2.3 Final Speedup Computation

With the values of $p_{i,j}$ computed as described in Section 7.2.2, now $p(Tpar_{tuple_i} = W_j)$ can be computed, the probability that the parallel time of a tuple i is equal to some thread size $W_j \in E$. This term can be computed as:

$$p(Tpar_{tuple_i} = W_j) = \prod_{k=0}^{P-1} \left(\sum_{l=1}^{rank(j)} p_{rank(l),k} \right) - \sum_{m=1}^{rank(j)-1} p(Tpar_{tuple_{i'}} = W_{rank(m)}) \tag{7.12}$$

where $rank(x)$ is a function that maps the indexes used to name threads, e.g., i in W_i, to the rank of each thread in the ordered set E (Section 7.2.1) [1]. The first term in Equation 7.12 is simply the probability that the parallel time of a tuple is equal to any of the thread sizes up to $rank(j)$, inclusive. The second term in Equation 7.12 is then the probability that the parallel time of a tuple is equal to any of the thread sizes up to $rank(j-1)$. Note that Equation 7.12 can be computed recursively and there is no need to recompute the second term for each j. In the notation for summations it is assumed that when the upper bound is less than the lower bound the sum defaults to a value of zero. So, the second term defaults to zero when $j = 1$. Equation 7.12 is an extended version of Equation 5.5.

[1] This naming indirection is necessary because the overhead threads are named after all base threads have been named and ordered in set B.

With the results of Equation 7.12, Equation 7.6 can then be computed, and, finally, with this result and that of Equation 7.5, the estimated speedup can be computed as:

$$S_{est} = \frac{Tseq_{est}}{Tpar_{est}} \tag{7.13}$$

Section 7.6 will show an example of how to use the above equations. Finally, note that when the number of loop iterations is known to be less than the number of processors in the system, the value of P in the formulas above will be replaced with the number of iterations.

7.3 Overhead Thread Sizes

The previous section introduced the extended thread tuple model and assumed that all thread sizes, base and overhead, had been created and ordered in the B and E sets. The computations of the previous section also assumed that the probabilities of occurrence of some overheads, such as p_{dep} in Equation 7.8 and p_{ovflow} in Equation 7.9, had also been pre-determined. This section explains in detail how the thread sizes with overheads are created starting from the base thread sizes (whose generation is explained in Section 5.2).

7.3.1 Initialisation

The tuple model relies on the estimation of the possible thread sizes and their probabilities of occurrence. The starting point is then to compute the thread sizes that originate from the different execution paths, the base thread sizes, which are then stored in a *base thread table*. Figure 7.1a shows a skeleton of a loop that is being considered for speculative parallelisation and Figure 7.1b shows the two base thread sizes due to the two possible execution paths with their probabilities of occurrence. Section 5.2 described in more detail how to compute the base thread sizes and their probabilities.

The extended tuple model extends the base tuple model by adding the overhead thread sizes and their probabilities, which are then stored in an *extended thread table*. Since in the extended tuple model threads have different probabilities of occurrence on different processor slots it is necessary to extend the base thread table to have multiple entries for the probabilities. As the overhead thread sizes are derived from the original

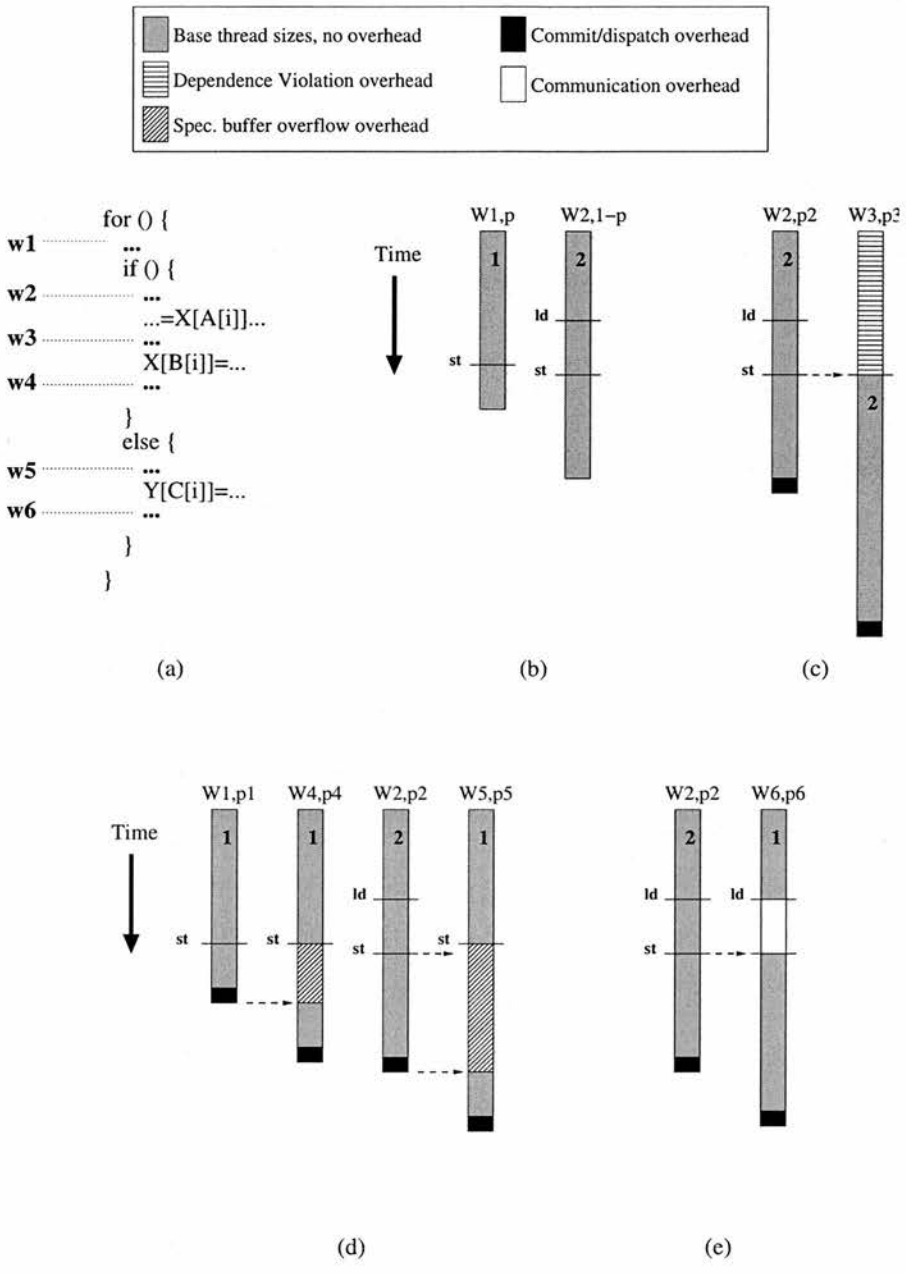

Figure 7.1: Example of modelling speculative parallelisation overheads in the extended tuple model: skeleton code to be speculatively parallelised (a); base thread sizes and probabilities from the two possible execution paths (b); additional thread sizes and probabilities when overheads are included (c) (d) (e). The numbers inside the bars identify the base thread sizes.

base thread sizes, the original probabilities of occurrence of the base threads are divided into two separate probabilities: the overhead thread's probabilities of occurrence and the base non-overhead thread size's probabilities of occurrence. The sum of these two terms must be the original probability of execution of the base thread size.

To simplify the problem, and to comply to the mathematical model of Section 7.2 a restriction that a base thread size can only possibly suffer from one source of overhead is imposed. Thus, a base thread size cannot be squashed and at the same time overflow the speculative write buffer and stall. The only exception is dispatch and commit overheads, which can be added to all thread sizes, including overhead ones, as described in Section 7.3.5. Another restriction is that an extended thread size cannot generate any further thread sizes by either squashing threads or by forcing stalled successors to wait. Note that these restrictions imply that $N < M^2$, where N is the total number of thread sizes considered by the model and M is the number of base thread sizes. In case some base thread size may suffer from more than one source of overhead then one overhead must be chosen. For this purpose it is proposed to choose to model only the overhead that would appear first: if the squashing store in the producer predecessor is expected to appear before the stalling store in the thread itself, then consider that the thread is only squashed, and vice-versa. Another option would be to consider the overhead that occurs with the greatest probability.

Following notation is used for the timestamps. The time stamp of the squash is the time stamp of the store operation from the producer, T_{prod_str}, whereas the time stamp of an speculative buffer overflow is determined by the time stamp of the store operation that causes the stall, $T_{stalling\_str}$.

7.3.2 Squash and Restart Overhead

Since the major contribution in the squash and restart overhead is the re-execution of part of the thread after being squashed, the final size of the squashed thread is the original thread size plus the execution time of the predecessor producer thread until the violation is detected, which is usually shortly after the store involved in the RAW violation [2]. Thus, if the time stamp of the producer store is T_{prod_str} then the size of a squashed thread W_i can be computed as:

[2] In case the squashed thread has already finished and is waiting to commit by the time the violation is detected, the idle time is attributed to squash overhead and not load imbalance overhead.

$$W_i = W_{base} + T_{prod\_str} \tag{7.14}$$

where W_{base} is the original base thread size. Figure 7.1c shows the case of a potential data dependence violation and squash, in which case the probability of the original base thread size $W2$ is adjusted and a new thread size is created for the squashed case, $W3$.

The probabilities of occurrence of this squashed thread size and of the original base thread size vary depending on the relative position of the thread in the tuple and are computed through Equation 7.8. The rationale behind this equation is the following. In order for a squashed thread size to appear in a given processor slot it is necessary that three conditions be true: (1) the base thread appears in this processor slot; (2) the producer thread is present in at least one of its predecessor processor slots; and (3) the data dependence does occur in this particular execution instance of the producer and consumer threads. The probability of the squashed thread is then the joint probability of these events (which are considered to be independent). The term $(1-(1-p_{prod})^j)$ in Equation 7.8 is the reverse probability that no producer thread appears in any predecessor processor slot. This is, in other words, the probability that a producer thread is present in at least one of its predecessor processor slots. The term p_{base} in Equation 7.8 is the probability that the base thread appears in a particular processor slot. Finally, the term p_{dep} in Equation 7.8 is the probability that the data dependence does occur [3]. Note that the result of Equation 7.8 is zero for a squashed thread size and processor zero, which is to be expected since no squashed thread size should appear on the non-speculative processor.

In order for the original base, non-squashed, thread size to appear in a given processor slot it is necessary that the base thread appears in this processor slot and that one of two conditions occur: (1) the producer thread is not present in any predecessor processor slot or (2) if that is not the case, the data dependence does not occur. The term $(1-p_{prod})^j$ in Equation 7.8 is the probability that no producer thread appears in any predecessor processor slot and corresponds to the first condition. The term $(1-(1-p_{prod})^j)$ in Equation 7.8 is the probability that a producer thread is present in

[3] Obviously, if the data dependence between the producer store and the consumer load can be determined at compile time to be true every time these threads execute, then p_{dep} is equal to one. In many cases, however, the dependence information is incomplete, such as when the memory operations are through subscripted subscripts or through pointers.

at least one of its predecessor processor slots and the term $(1 - p_{dep})$ is the probability that the data dependence does not occur. Thus, combined, these two terms correspond to the second combination of conditions. Note that the result of Equation 7.8 is simply p_{base} for a non-squashed thread size and processor zero, which is to be expected since only the non-squashed size should appear on a non-speculative processor.

This model and Equation 7.8 assume that a base thread can only be squashed by a single producer. It may occur, however, that a consuming load may be dependent on producing stores from more than one different control paths, i.e., from different base threads [4]. In this case the model is extended as follows.

The producer stores from the different threads are combined into a single proxy producer store and, similarly, the producer threads are combined into a single proxy producer thread. More specifically, if $T_{prod\_str_i}$ is the time stamp of the producer store in the producer base thread W_i then the time stamp of the proxy producer store is the weighted average of all producer stores and Equation 7.14 becomes:

$$W_j = W_{base} + mean(T_{prod\_str}) \tag{7.15}$$

where W_{base} is the original base thread size, W_j is the squashed thread size, and $mean(T_{prod\_str})$ is the weighted mean of all the producer store timestamps weighted by the probabilities of occurrence of their threads. The latter is defined as:

$$mean(T_{prod\_str}) = \sum_{W_i \in PROD} T_{prod\_str_i} \cdot Weight_{W_i} \tag{7.16}$$

where $PROD$ is the set of producer threads and $Weight_{W_i}$ is defined as:

$$Weight_{W_i} = \frac{p_i}{p_{producer_{combined}}} \tag{7.17}$$

where $p_{producer_{combined}}$ is the probability of the proxy producer thread and is the sum of the probabilities of all producer threads:

$$p_{producer_{combined}} = \sum_{W_i \in PROD} p_i \tag{7.18}$$

Similarly, the probability of dependence is also the weighted mean of all the probabilities of dependence weighted by the probabilities of occurrence of their threads:

[4] Note that this case is different from the case described in Section 7.3.1 where multiple loads in a thread can cause violations. This case is also different from the case where a single producer thread may have multiple producing stores, in which case only the earliest store is considered.

$$mean(p_{dep}) = \sum_{W_i \in PROD} p_{dep_i} \cdot Weight_{W_i} \tag{7.19}$$

where p_{dep_i} is the probability of dependence between the producing store in the producer thread W_i and the consuming load and $Weight_{W_i}$ is as defined above.

Finally, the term $p_{producer_{combined}}$ replaces the term p_{prod} and the term $mean(p_{dep})$ replaces the term p_{dep} in Equation 7.8.

Section 7.4.2 contains the algorithm for computing the squash and restart overhead.

The methodology described above to model squash and restart overheads does not take into account the fact that at a violation not only the consumer thread but also all its successors must be squashed. This inaccuracy is likely to have more significant impact in systems with large number of processors. In this thesis, the decision was made to trade off accuracy for simplicity of the model.

One of the major difficulties of estimating data dependence violations under speculative parallelisation is that by construction these are highly unpredictable (more predictable data dependences are more efficiently handled by explicit synchronisation [10, 30, 44]). Estimating the likelihood of violations can be done with static probabilistic memory disambiguation analysis [8, 53] or profiling [29, 31].

7.3.3 Speculative Buffer Overflow Overhead

The major contribution in the speculative buffer overflow overhead is the idle time the thread spends waiting for all its predecessor threads to commit, which is approximately the idle time waiting for its longest predecessor to complete execution. Thus, the size of a thread that stalls due to speculative buffer overflow is equal to the execution time of the predecessor thread plus the execution time remaining to complete execution once the thread is released. If the time stamp of the store that causes the stall is T_{stall_str} then the size of the overflowed thread W_i can be computed as:

$$W_i = W_{waitfor} + (W_{base} - T_{stall\_str}) \tag{7.20}$$

where W_{base} is the original base thread size and $W_{waitfor}$ is the longest predecessor W_i has to wait for. In practise several thread sizes could assume this role of longest predecessor, as long as their execution time is longer than the time stamp of the stalling store. To keep the model simple only predecessor thread sizes that are base thread sizes

are considered. Thus, in the model, from the original thread size (the one suffering the speculative buffer overflow), we can generate as many new thread sizes as there are base thread sizes with execution times longer than the time stamp of the stalling store (note that this includes the original base thread size itself). Figure 7.1d shows the case of a potential speculative buffer overflow of thread size $W1$, in which case it must wait for either another instance of $W1$ or an instance of $W2$ generating $W4$ or $W5$, respectively.

The probabilities of occurrence of an overflowed thread size and of the original base thread size vary depending on the relative position of the thread in the tuple and on the ranking of the stalling predecessor thread in the base thread table and are computed through Equation 7.9. The rationale behind this equation is the following. In order for an overflowed thread size corresponding to a certain stalling predecessor thread to appear in a given processor slot it is necessary that four conditions be true: (1) the base thread appears in this processor slot; (2) the stalling thread is present in at least one of its predecessor processor slots; (3) no thread longer than the stalling thread appears in any predecessor processor slot; and (4) the overflow does occur in this particular execution instance of the base thread. The probability of an overflowed thread is then the joint probability of these events (which are assumed to be independent). The term $(\sum_{k=1}^{waitfor} p_k)^j$ in Equation 7.9 is the probability that only base thread sizes in the range $k = 1$ to $k = waitfor$ appear in all of its predecessor processor slots. Similarly, the term $(\sum_{k=1}^{waitfor-1} p_k)^j$ in Equation 7.9 is the probability that only base thread sizes in the range $k = 1$ to $k = waitfor - 1$ appear in all of its predecessor processor slots. Thus the difference of the two terms corresponds to conditions (2) and (3). The term p_{base} in Equation 7.9 is the probability that the base thread appears in a particular processor slot. Finally, the term p_{ovflow} in Equation 7.8 is the probability that the overflow does occur [5]. Note that the result of Equation 7.9 is zero for an overflowed thread size and processor zero, which is to be expected since the non-speculative processor will never stall.

In order for the original base, non-overflowed, thread size to appear in a given processor slot it is necessary that the base thread appears in this processor slot and that one of two conditions occur: (1) the overflow does not occur or (2) if that is not the

[5] Obviously, if the overflow can be determined at compile time to be true every time the base thread executes, then p_{ovflow} is equal to one. In many cases, however, the overflow information is incomplete, such as when the mapping of data to cache lines and cache sets cannot be resolved at compile time.

case, no thread size with running time larger than the time stamp of the stalling store appears in any of the predecessor processor slots. The term p_{base} in Equation 7.9 is the probability that the base thread appears in a particular processor slot. The term $1 - p_{ovflow}$ in Equation 7.9 is the probability that the overflow does not occur and corresponds to the first condition. The term $(1 - \sum_{k=firstlonger}^{|B|} p_k)^j$ is the probability that no thread size between $k = firstlonger$ to $k = |B|$ appears in any of the predecessor processor slots and corresponds to the second condition. Note that the result of Equation 7.9 is simply p_{base} for a non-overflowed thread size and processor zero, which is to be expected since only the non-overflowed size should appear on a non-speculative processor.

As a new overflowed thread size is generated for each base thread size $W_{waitfor}$ in the range $firstlonger \leq waitfor \leq |B|$, after generating all the overflowed thread sizes the worst case number of the overflowed thread sizes is $|B|^2$, when base thread size overflows with $firstlonger = 1$. If compilation time and memory resources are limited it is possible to combine all thread sizes in the $firstlonger \leq waitfor \leq |B|$ range into a single thread size. More specifically, if T_{stall_str} is the time stamp of the stalling store in the base thread size W_{base} then Equation 7.20 becomes:

$$W_j = mean(W_{waitfor}) + (W_{base} - T_{stall\_str}) \tag{7.21}$$

where W_j is the (single) overflowed thread size and $mean(W_{waitfor})$ is the weighted mean of all thread sizes in the range $firstlonger \leq waitfor \leq |B|$ weighted by the probabilities of occurrence of their threads. The mean is defined as:

$$mean(W_{waitfor}) = \sum_{i=firstlonger}^{|B|} W_i \cdot Weight_{W_i} \tag{7.22}$$

where $Weight_{W_i}$ is defined as:

$$Weight_{W_i} = \frac{p_i}{p_{waitfor_{combined}}} \tag{7.23}$$

where $p_{waitfor_{combined}}$ is the probability of the combined longest predecessor thread and is the sum of the probabilities of all producer threads:

$$p_{waitfor_{combined}} = \sum_{i=firstlonger}^{|B|} p_i \tag{7.24}$$

Finally, Equation 7.9 becomes:

$$p_{i,j} = \begin{cases} (1 - p_{ovflow}) \cdot p_{base} + p_{base} \cdot p_{ovflow} \cdot (1 - p_{waitfor_{combined}})^j \\ \quad \text{, if } W_i \text{ is a base size} \\ (1 - (\sum_{k=1}^{firstlonger-1} p_k)^j) \cdot p_{ovflow} \cdot p_{base} \\ \quad \text{, if } W_i \text{ is an overflowed size} \end{cases} \tag{7.25}$$

With this optimisation only one overflowed thread size is generated for each base thread size that overflows the speculative buffer, so in the worst case the total number of overflowed thread sizes is only $|B|$.

Section 7.4.2 contains the algorithm for computing the squash and restart overhead.

Determining the probability of a speculative buffer overflow is also a very difficult problem. In systems that have dedicated fully-associative speculative buffers, as in [19], this problem reverts to the problem of counting the number of speculative buffer lines covered by the stores in the execution path under consideration. In systems that have only the set-associative L1 caches, as in [41], the problem is that of finding whether two memory accesses conflict in the cache. This problem has been well addressed for regular applications (e.g., [6, 47]), but is still an open problem for irregular applications.

7.3.4 Inter-thread Communication Overhead

Some systems support, in addition to data speculation, explicit synchronised communication through memory or registers [10, 23, 30, 41, 44]. This can be modelled as additional execution time added to the base thread between the time of the consumption and the production of the data. Thus, the size of a thread that stalls due to synchronised inter-thread communication is its original execution time plus the time between the stalling load and the releasing store. If the time stamp of the load that causes the synchronisation stall is T_{stall_ld} and the time stamp of the store that releases the stalled thread is T_{rel_str} then the size of the synchronised thread W_i can be computed as:

$$W_i = W_{base} + (T_{rel\_str} - T_{stall\_ld}) \tag{7.26}$$

where W_{base} is the original base thread size. Figure 7.1e shows the case of a potential synchronisation and inter-thread communication, in which case the potential memory dependence on array X is handled by explicit synchronisation and the probability of the original base thread size $W2$ is adjusted and a new thread size is created for the synchronised case, $W6$.

Two approaches for synchronised inter-thread communication have been proposed. In one approach the synchronisation is placed dynamically at run time [10, 30, 44] and is usually triggered only when a dependence does occur, or is thought to occur (for example by triggering synchronisation depending on memory addresses [10]). In this case the probabilities of occurrence of the synchronised thread size and of the original base thread size vary depending on the relative position of the thread in the tuple and are computed through Equation 7.10. The rationale behind this equation is analogous to that of Equation 7.8 explained in Section 7.3.2 and is not repeated here for the sake of brevity.

In the second approach the synchronisation is placed statically [23, 41] and is usually triggered each time the load appears (for example by associating the synchronisation with the static loads in the source code or with accesses to "empty" registers). In this case the probabilities of occurrence of the synchronised and of the original base thread size simply depend on whether the position in the tuple is processor slot zero or some other processor slot, and are computed through Equation 7.11. The rationale behind this equation is the following. The synchronised thread size cannot appear in the non-speculative processor slot and will appear each time the original base thread size appears in any other processor slot. The original thread size, on the other hand, cannot appear in any speculative processor slot and will not be replaced when it appears in the non-speculative processor slot. Thus, the probability that the original base thread size (synchronised thread size) appears in processor zero is p_{base} (zero) and in the other processor slots is zero (p_{base}).

7.3.5 Dispatch and Commit Overheads

Different from the other overheads, thread dispatch and commit overhead is not represented as additional thread sizes in the model. Instead, the time required by these operations is added to the execution time of the existing thread sizes. However, due to the in-order commit requirement, the amount of dispatch and commit overhead ob-

served by a thread is not exactly the nominal time required by these operations, but depends on the position of the longest thread in the tuple. Simply adding the nominal cost of the operations to the thread sizes would incorrectly overlap some of the commit costs. In reality, when the longest thread is the non-speculative one, the total overhead is equal to P times the nominal dispatch and commit time, and when the longest thread is the most-speculative one, the total overhead is simply equal to the nominal dispatch and commit time. In the average case the total overhead is equal to $T_{disp/comm} \cdot (\sum_{j=1}^{P} j)/P = T_{disp/comm} \cdot (P+1)/2$, where $T_{disp/comm}$ is the nominal dispatch and commit time, which is assumed to be constant for simplicity. Figure 7.1 shows the commit and dispatch overheads added to all the thread sizes.

The thread dispatch time is usually a somewhat fixed time required to update some system data structures, possibly copy-in some register values, and dispatch the new thread on the processor. The commit time, however, is more variable as it depends on the amount of data that is modified by the thread while it is speculative. The analysis required to compute this is very similar to that required to estimate speculative buffer overflows. This overhead also depends on the contention at the bus during the write-backs of data from the speculative buffers. Some speculative multithreaded systems support "lazy commit", whereby data modified by a non-speculative thread may reside in the speculative buffer after the processor completes its execution and starts work on a more speculative thread [16]. This data is then lazily written back to memory on demand when more space in the speculative buffer is required by the new thread. In this way the commit overhead is spread over time and can be somewhat hidden. Such commit mechanism, however, requires additional costly hardware and is not supported by most speculative CMP's.

7.4 Implementation Issues

7.4.1 memory access operations (*memop*)

During the CCFG traversal, *memops* collected from the basic block nodes that contain them. *memops* (memory operations) stores the relevant information about the load and store operations. Its definition is presented in Figure 7.2. They are essential for computing the overhead thread sizes for the extended thread table.

```
struct memop {
    address,
    time stamp,
    access type (load/store),
    probability,
    parent thread ID,
    }
```

Figure 7.2: *memop* interface

In the presence of an inner loop which has an iteration count (n) greater than one, the *memops* of the inner loop are repeatedly accessed n times by each iteration. This behaviour is captured in the model by creating a new version of *memop* for each iteration. The time stamps of the *memops* of are computed by adding the sum of all of the previous iterations' workload: $T_{memop}(i) = T_{memop}(1) + (i-1) \cdot W_{iteration}$, where $T_{memop}(i)$ is the time stamp for the created version of *memop* for iteration i; $W_{iteration}$ is the estimated workload for the inner loop, and $i \in [2,n]$.

7.4.2 Algorithm for Generating the Overhead Thread Sizes

The algorithm for generating the extended thread table from base thread table is shown in figure 7.3. This algorithm is based on the tuple model described in section 7.2 and 7.3.

```
Step 1: initialise p_{i,j}
for i ∈ [1,|B|]:
   for j ∈ [0,P]
      p_{i,j} = p_i
Step 2: squash and restart overheads:
for each pair of data dependent memops
   create new table entry using Equation 7.14, 7.8
if there are more than one producer threads for a consumer
   merge them using Equation 7.16, 7.18
update base thread table entry using Equation 7.8.
Step 3: speculative buffer overflow overheads:
for each entry k in the base thread table:
   if k exceeds the estimated buffer capacity:
      determine firstlonger for k
#if NO_OPTIMISATION
      for longestpred in [firstlonger,|B|]:
         create new table entry using entry Equation 7.20, 7.9
         update base thread table entry using Equation 7.9.
#else
      merge threads [firstlonger,|B|] into one longestpred
      create new table entry using entry Equation 7.20, 7.9
      update base thread table entry using Equation 7.9.
#endif
  Step 4: add dispatch and commit overhead
  add the estimated dispatch and commit overhead to each table entry
  Step 5: sort table
  sort table in increasing order of thread sizes
```

Figure 7.3: Algorithm for building the extended thread table from the base thread table

7.5 Algorithm for Generating the Speedup from the Extended Thread Size Table

The algorithm shown in figure 7.4 generates the value of speedup from the extended thread size table. This algorithm corresponds to Equation 7.5, Equation 7.6, Equation 7.12 and Equation 7.13 in Chapter 7. The input for this algorithm is the extended thread size table and the output is the speedup. In this algorithm, each $Ppar_{\sum}$ represents a single value, which is the sum of Ppar for a certain number of table entries; $p_{i,j}$ represents a single value, which is the probability of a thread size from a given table entry i on a given processor j; $P_{\sum i,j}$ represents a set of values, where the number of elements in this set is the number of processor, P, and each value in this set is the the sum of $p_{i,j}$ on a given processor; $P_{\sum i,\prod j}$ represent a single value, which is the product of $P_{\sum i,j}$ for each $j \in P$. By saving intermediate results, this algorithm has a computational complexity of $O(N \cdot P)$, where N is the number of entries in the extended thread size table.

```
initialise Ppar_Σ to 0
for each j ∈ P, initialise P_Σi,j to 0
for each entry i in the extended thread size table:
   initialise P_Σi,∏j to 1
   if i is base thread size entry:
      Tseq_est =+ i.size() · i.Pseq
   for each PE j ∈ P:
      p_Σi,j =+ i.p_i,j
      P_Σi,∏j =* P_Σi,j
   i.Ppar = P_Σi,∏j - Ppar_Σ
   Tpar_est =+ i.size()· i.Ppar
   Ppar_Σ =+ i.Ppar
speedup = Tseq_est / Tpar_est
```

Figure 7.4: Algorithm for Generating Speedup from the extended thread table

7.5.1 Complexity Analysis

The complexity analysis of the proposed framework is composed of three phases: the generation of all the base thread sizes (base thread table); the generation of the overhead thread sizes (extended thread table); and the computation of the estimated speedup. They are shown in Table 7.2.

It is broken down as follows. From the CCFG representation it is possible to generate all the base thread sizes in $O(M)$, where M is the number of control paths in the program, if information is carried and stored at intermediate nodes to avoid traversing any path segment more than once. After generating all the base thread sizes, the sorting of the base table has a complexity of $O(M \cdot \log_2 M)$.

Phase	*Equation (s)*	*Complexity*
CCFG $\rightarrow$ base thread table	N/A	$O(M + M \cdot \log_2 M)$
Base thread table $\rightarrow$ extended thread table	See Table 7.3	$O(M^2 + M \cdot P + R \cdot \log_2 R)$
Extended thread table $\rightarrow$ the estimated speedup	7.6, 7.5, 7.12, 7.13	$O(N \cdot P + N \cdot \log_2 N)$
Summary		$O(M^2 + N \cdot \log_2 N + N \cdot P)$ $+R \cdot \log_2 R)$

Table 7.2: Complexity for the model

The complexity of the generation of the overhead thread sizes varies in accordence with the overhead. They are shown in Table 7.3. They are broken down as follows. In the squashed and restart overhead, a naive search for possible data dependence violations would require comparing the target and timestamps of each memory reference in a base thread against all memory references in all other threads, which would lead to a squared complexity. However, by storing all the memory references in a hash table it is possible to reduce the *expected* complexity to $O(R \cdot \log_2 R)$, where R is the total number of memory references collected during the CCFG traversal. If multiple producers described in Section 7.3.2 is supported, merging the producers has a complexity of $O(M^2)$. Thus the computation of the squashed thread sizes has a complexity of $O(R \cdot \log_2 R + M^2)$. In the speculative buffer overflow overhead, searching for possible speculative buffer overflow occurrences requires considering all memory

references for each base thread, which leads to $O(R)$. The generation of the overflowed thread sizes has the complexity of $O(M^2)$. Thus the overall complexity for the overflowed thread sizes is $O(R+M^2)$. The complexity of the communication overhead is the same as the squashed overhead. Finally, Handling the dispatch and commit overhead involves adding a fixed time to each thread size in the extended thread table, thus $O(N)$. So, the overall computation complexity of generating the overhead thread sizes is $O(M^2+M\cdot P+R\cdot \log_2 R)$.

Phase	***Equation (s)***	***Complexity***
Initialisation	7.7	$O(M\cdot P)$
Squash and Restart	7.8, 7.14, 7.15, 7.16 7.8, 7.14, 7.15, 7.16	$O(R\cdot \log_2 R+M^2)$
Buffer Overflow	7.9, 7.20	$O(R+M^2)$
Communication	7.10 7.26	$O(R\cdot \log_2 R+M^2)$
Dispatch and Commit	N/A	$O(N)$
Summary		$O(M^2+N\cdot \log_2 N+M\cdot P+R\cdot \log_2 R)$

Table 7.3: Complexity for Computing the Overhead Thread Sizes

By saving the sum of the probabilities for the intermediate results, as the algorithm shown in Figure 7.4, the computation of the estimated speedup has the computational complexity of $O(N\cdot P)$. After all the overhead thread sizes are generated, the sorting of the extended table has a complexity of $O(N\cdot \log_2 N)$

Thus, the overall computation complexity of the proposed framework is $O(M^2+N\cdot \log_2 N+N\cdot P+R\cdot \log_2 R)$.

7.6 An Example Computation of S_{est} with the Extended Model

As an example, consider the code section in Figure 7.1a. The corresponding base thread table is shown in Table 7.4. Thus, $B = \{W_1, W_2\}$. Assume a system with four processors.

Thread name	*Probability of occurrence (p_i)*	*Execution time*	*Memory operations*	*Example parameters*
W_1	50%	$w1+w5+w6$	st_Y@T_1	$W = 100; T_1 = 50$
W_2	50%	$w1+w2+w3+w4$	ld_X@T_2	$W = 200$
			st_X@T_3	$T_2 = 30; T_3 = 120$

Table 7.4: Base thread sizes, probabilities, and memory operations with timestamps (i.e., base thread table) for the example in Figure 7.1a. The thread sizes are sorted in increasing order (thus $w2+w3+w4 > w5+w6$). The values in the last column are arbitrary and only for the sake of the example.

Looking at the memory operations associated with W_2 we can assume the possibility of a data dependence violation when two different *instances* of this thread size appear in the same tuple and the address of the store of the predecessor is the same as the address of the load of the successor. This leads to a new possible thread size W_3, as shown in Table 7.5 and Figure 7.1c. For the sake of the example, assume that the probability that the same memory addresses are accessed by two instances of W_2 is $p_{dep} = 0.1$.

Also, looking at the memory operations associated with W_1 we can assume the possibility of a speculative buffer overflow at the store of this thread. This leads to two new possible thread sizes W_4, when the predecessor to wait for is an instantiation of W_1, and W_5, when the predecessor to wait for is an instantiation of W_2, as shown in Table 7.5 and Figure 7.1d. For the sake of the example, assume that the probability that the store will overflow the speculative buffer is $p_{ovflow} = 0.3$. Thus, $E = \{W_1, W_4, W_2, W_5, W_3\}$.

The next step is to compute the terms $p_{i,j}$, the probabilities that thread size W_i appears in processor j in a given tuple. The terms for $i = 2$ and $i = 3$ fall in case 2 of Section 7.2.2, as W_2 is associated with a squash overhead. Thus, using Equation 7.8

Thread name	Final rank	*Execution time*	*Example size*
W_1	1	$w1+w5+w6$	100
W_2	3	$w1+w2+w3+w4$	200
W_3	5	$2*(w1+w2+w3)+w4$	320
W_4	2	$(w1+w5+w6)+w6$	150
W_5	4	$(w1+w2+w3+w4)+w6$	250

Table 7.5: Base and extended thread sizes (i.e., extended thread table) for the example in Figure 7.1a.

and substituting both p_{prod} and p_{base} with p_2:

$$
\begin{aligned}
p_{2,0} &= (1-p_2)^0 * p_2 + (1-(1-p_2)^0) * p_2 * (1-p_{dep}) = 0.5 \\
p_{2,1} &= (1-p_2)^1 * p_2 + (1-(1-p_2)^1) * p_2 * (1-p_{dep}) = 0.475 \\
p_{2,2} &= (1-p_2)^2 * p_2 + (1-(1-p_2)^2) * p_2 * (1-p_{dep}) = 0.4625 \\
p_{2,3} &= (1-p_2)^3 * p_2 + (1-(1-p_2)^3) * p_2 * (1-p_{dep}) = 0.45625
\end{aligned}
$$

$$
\begin{aligned}
p_{3,0} &= (1-(1-p_2)^0) * p_2 * p_{dep} = 0 \\
p_{3,1} &= (1-(1-p_2)^1) * p_2 * p_{dep} = 0.025 \\
p_{3,2} &= (1-(1-p_2)^2) * p_2 * p_{dep} = 0.0375 \\
p_{3,3} &= (1-(1-p_2)^3) * p_2 * p_{dep} = 0.04375
\end{aligned}
$$

Note that for every j the sum of $p_{2,j}$ and $p_{3,j}$ equals $p_2 = 0.5$, as expected. Also note that for increasing j the value of $p_{2,j}$ decreases and the value of $p_{3,j}$ increases. This is also expected as the chances of a data dependence violation increases with more predecessors. This behaviour is not captured in the base model of Part II and in any of the current compiler models.

The terms for $i = 1$, $i = 4$, and $i = 5$ fall in case 3 of Section 7.2.2, as W_1 is associated with a speculative buffer overflow overhead. Thus, using Equation 7.9 and substituting p_{base} with p_1, $p_{firstlonger}$ with p_1, $p_{waitfor}$ with p_1 for W_4, and $p_{waitfor}$ with

p_2 for W_5:

$$
\begin{aligned}
p_{1,0} &= p_1 * (1 - p_{ovflow}) + p_1 * p_{ovflow} * (1 - \sum_{k=1}^{2} p_k)^0 = 0.5 \\
p_{1,1} &= p_1 * (1 - p_{ovflow}) + p_1 * p_{ovflow} * (1 - \sum_{k=1}^{2} p_k)^1 = 0.35 \\
p_{1,2} &= p_1 * (1 - p_{ovflow}) + p_1 * p_{ovflow} * (1 - \sum_{k=1}^{2} p_k)^2 = 0.35 \\
p_{1,3} &= p_1 * (1 - p_{ovflow}) + p_1 * p_{ovflow} * (1 - \sum_{k=1}^{2} p_k)^3 = 0.35
\end{aligned}
$$

$$
\begin{aligned}
p_{4,0} &= ((\sum_{k=1}^{1} p_k)^0 - (\sum_{k=1}^{0} p_k)^0) * p_{ovflow} * p_2 = 0 \\
p_{4,1} &= ((\sum_{k=1}^{1} p_k)^1 - (\sum_{k=1}^{0} p_k)^1) * p_{ovflow} * p_2 = 0.075 \\
p_{4,2} &= ((\sum_{k=1}^{1} p_k)^2 - (\sum_{k=1}^{0} p_k)^2) * p_{ovflow} * p_2 = 0.0375 \\
p_{4,3} &= ((\sum_{k=1}^{1} p_k)^3 - (\sum_{k=1}^{0} p_k)^3) * p_{ovflow} * p_2 = 0.01875
\end{aligned}
$$

$$
\begin{aligned}
p_{5,0} &= ((\sum_{k=1}^{2} p_k)^0 - (\sum_{k=1}^{1} p_k)^0) * p_{ovflow} * p_2 = 0 \\
p_{5,1} &= ((\sum_{k=1}^{2} p_k)^1 - (\sum_{k=1}^{1} p_k)^1) * p_{ovflow} * p_2 = 0.075 \\
p_{5,2} &= ((\sum_{k=1}^{2} p_k)^2 - (\sum_{k=1}^{1} p_k)^2) * p_{ovflow} * p_2 = 0.1125 \\
p_{5,3} &= ((\sum_{k=1}^{2} p_k)^3 - (\sum_{k=1}^{1} p_k)^3) * p_{ovflow} * p_2 = 0.13125
\end{aligned}
$$

Note that for every j the sum of $p_{1,j}$, $p_{4,j}$, and $p_{5,j}$ equals $p_1 = 0.5$, as expected. Also note that for every j $p_{4,j} \leq p_{5,j}$. This is also expected as for W_4 to appear it is necessary that W_2 does not appear in any predecessor processor slot, while W_5 can appear even if W_1 appears in several predecessor processor slots, as long as W_2 appears

in at least one predecessor processor slot.

Now using Equation 7.12 the probabilities of the parallel execution time of a given thread tuple i (the correspondence between $W_{rank(j)}$ and some W_k are given in Table 7.5) can be computed:

$$p(Tpar_{tuple_i} = W_{rank(1)}) = \prod_{k=0}^{3}(\sum_{l=1}^{1} p_{rank(l),k}) - \sum_{m=1}^{0} p(Tpar_{tuple_{i'}} = W_{rank(m)}) = 0.0214$$

$$p(Tpar_{tuple_i} = W_{rank(2)}) = \prod_{k=0}^{3}(\sum_{l=1}^{2} p_{rank(l),k}) - \sum_{m=1}^{1} p(Tpar_{tuple_{i'}} = W_{rank(m)}) = 0.009$$

$$p(Tpar_{tuple_i} = W_{rank(3)}) = \prod_{k=0}^{3}(\sum_{l=1}^{3} p_{rank(l),k}) - \sum_{m=1}^{2} p(Tpar_{tuple_{i'}} = W_{rank(m)}) = 0.6007$$

$$p(Tpar_{tuple_i} = W_{rank(4)}) = \prod_{k=0}^{3}(\sum_{l=1}^{4} p_{rank(l),k}) - \sum_{m=1}^{3} p(Tpar_{tuple_{i'}} = W_{rank(m)}) = 0.2663$$

$$p(Tpar_{tuple_i} = W_{rank(5)}) = \prod_{k=0}^{3}(\sum_{l=1}^{5} p_{rank(l),k}) - \sum_{m=1}^{4} p(Tpar_{tuple_{i'}} = W_{rank(m)}) = 0.1026$$

Note that for every j the sum of $p(Tpar_{tuple_i} = W_j)$ equals 1, as expected. Also note that $p(Tpar_{tuple_i} = W_2)$ and $p(Tpar_{tuple_i} = W_5)$ are by far the largest terms. The first is expected since W_2 is a base size that is only replaced with a small probability (p_{dep}). The later occurs because, despite W_5 having smaller probabilities of appearing in any given processor than its corresponding base thread size, i.e. $p_{5,j} < p_{1,j}$ for every j, W_5 has a larger probability of deciding the parallel time since it is much larger than W_1. Finally, using Equations 7.5, 7.6, and 7.13, $Tseq_{est}$, $Tpar_{est}$, and S_{est} can be computed:

$$Tseq_{est} = 4 \cdot \sum_{i=1}^{2} W_i \cdot p_i = 600$$

$$Tpar_{est} = \sum_{j=1}^{5} W_j \cdot p(Tpar_{tuple_i} = W_j) = 223$$

$$S_{est} = \frac{Tseq_{est}}{Tpar_{est}} = \frac{600}{223} = 2.69$$

7.7 Limitations of the Model

One limitation of the model is that it dose not consider cascaded thread interactions. For instance, a thread may stall due to a speculative buffer overflow before it reaches the store that causes a data dependence violation with a successor thread; or a thread may stall due to a speculative buffer overflow and then wait for a predecessor that has also stalled due to a speculative buffer overflow. To model these the overhead models would have to be applied not only to base thread sizes but also to thread sizes that already incorporate some overheads. In practise this would make the model significantly more complex. Note also that there is the possibility of a combination on the same thread of a data dependence violation followed by a speculative buffer overflow when the thread is re-executed. There is no other possible combination except with dispatch and commit overhead [6]. Again, this compound overhead is not modelled to keep the model simple.

Another potential problem of the static compiler models is the fact that many parameters to the model may only be available at run time. For example, to more accurately compute the probabilities of certain thread sizes the model would require the probability of occurrence of the two directions in conditional statements. Also, to more accurately compute the thread sizes in the presence of inner loops the model would require the iteration count of these loops. A common approach to circumvent such lack of compile-time information is to use profiling. This model dose not necessarily resort to profiling, but any profiling required by the model could be done on a sequential execution of the program. Such profiling would then be much less expensive than full-blown profiling of the speculative multithreaded execution to directly determine the speedup of the speculative sections.

[6] In case the squashed thread has already finished and is waiting to commit by the time the violation is detected, the idle time is attributed to squash overhead and not load imbalance overhead. Also, in case a thread stalls due to overflow and is squashed before it can resume, the idle time is attributed to squash overhead and not overflow overhead.

Chapter 8

Evaluation

8.1 Speculative Parallelisation Performance

To assess how often speculative parallelisation leads to speedups or slowdowns with respect to sequential execution, Figure 8.1 shows, for each application, a histogram of the actual speedups obtained with the simulations for 4 processors. From this figure we observe that speculative parallelisation leads to a broad range of speedups and slowdowns: on average 56% of the loops lead to slowdowns while the other 44% lead to speedups. Note that this result is very similar to that of Section 8.1. These results evidence the importance of being able to selectively apply speculative parallelization.

8.2 Model Accuracy

8.2.1 Outcome (Speedup vs. Slowdown) Prediction Accuracy

The output of the compiler prediction model is an estimate of the actual value of the speedup or slowdown. The primary use of this prediction is to switch off speculative parallelisation of loops that are expected to lead to slowdowns and speculatively parallelise loops that are expected to lead to speedups. Figure 8.2 shows, for each application, a breakdown of the fraction of loops that are correctly predicted as leading to speedups (*Predict speedup/Actual speedup*), that are correctly predicted as leading to slowdowns (*Predict slowdown/Actual slowdown*), that are incorrectly predicted as leading to speedups (*Predict speedup/Actual slowdown*), and that are incorrectly predicted as leading to slowdowns (*Predict slowdown/Actual speedup*).

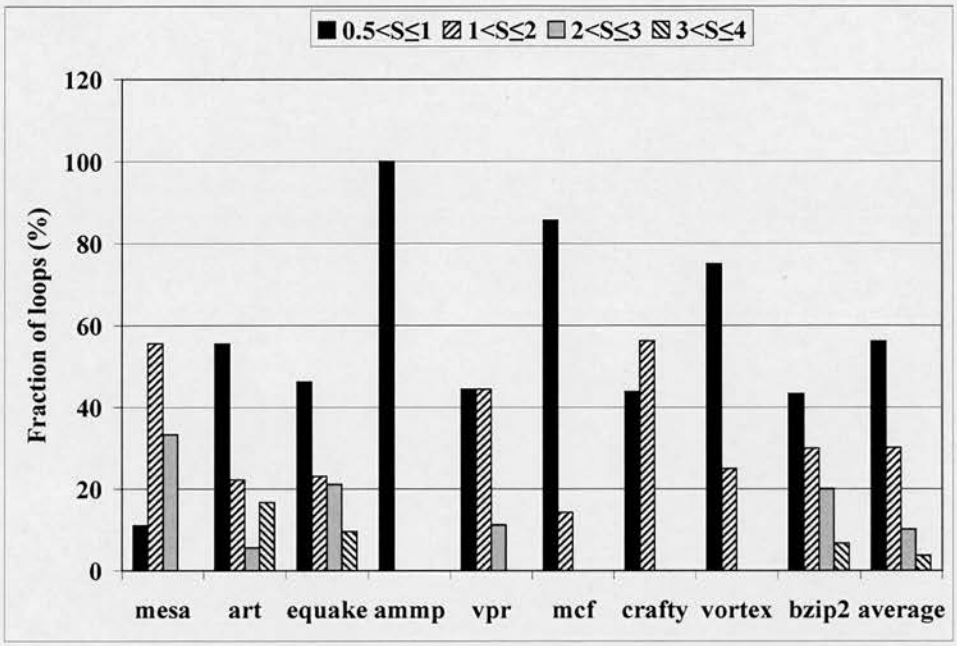

Figure 8.1: Histogram of speedups for the speculative loops only. Note that a speedup (S) less than one is a slowdown. In this experiment all loops except Doall loops are considered.

From this figure we can see that this model is accurate enough to correctly identify the speedup or slowdown outcome of the speculative execution in 67% of the cases, on average. Also, it seems that when the model is incorrect the tendency is to overestimate the performance gains. This means that the model rarely misses a valid opportunity to speculatively parallelize loops, but that it does not prevent all of the performance degradation from slowdown cases. Comparing with the results of Section 8.2, it has an increase in the *Predict speedup/Actual slowdown* at the expense of a decrease in the *Predict slowdown/Actual slowdown* category.

8.2.2 Dependence Violation Prediction Accuracy

When considering squash and restart overheads an important intermediate result of the compiler prediction model is its prediction of the occurrence or not of data dependence violations and, thus, squashes. Figure 8.3 shows, for each application, a breakdown

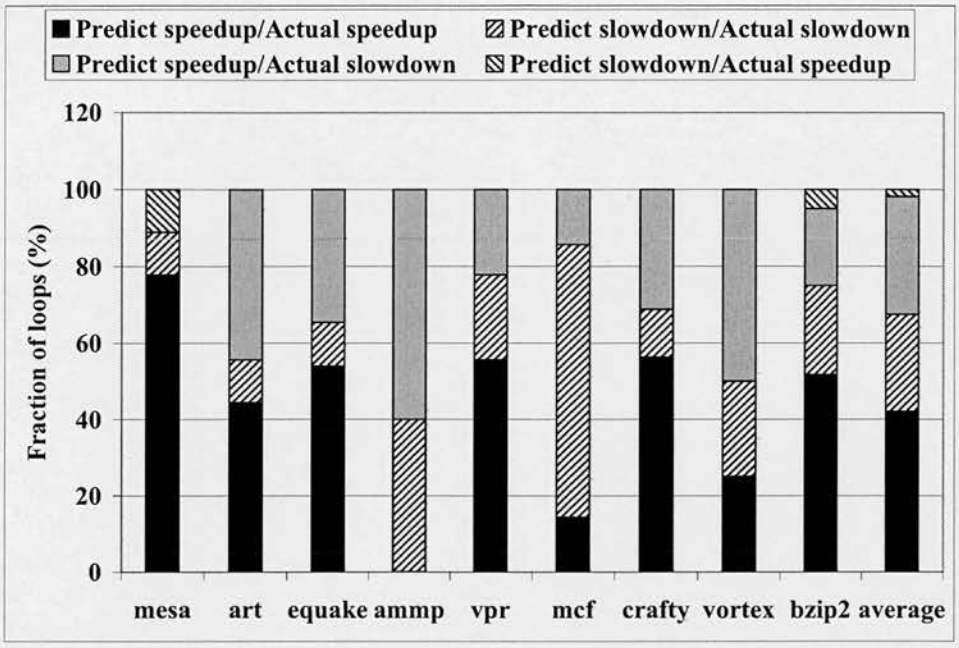

Figure 8.2: Outcome of speedup predictions with respect to actual observed speedup or slowdown. In this experiment all loops except Doall loops are considered.

of the fraction of loops that are correctly predicted as suffering violations (*Predict squash/Actual squash*), that are correctly predicted as not suffering violations (*Predict no squash/Actual no squash*), that are incorrectly predicted as suffering violations (*Predict squash/Actual no squash*), and that are incorrectly predicted as not suffering violations (*Predict no squash/Actual squash*). The third case, *Predict squash/Actual no squash*, occurs for two reasons: first, the compiler infrastructure does not yet take the indexes of arrays into consideration; and second, true data dependences may be hidden at run time if the statically computed timestamps are incorrect and the load actually appears after the store. The fourth case, *Predict no squash/Actual squash*, occurs only because the compiler infrastructure does not yet consider violations on scalar variables. Note that a loop is considered suffering violations as long as a violation occurs in at least one of the invocations of the loop.

This figure shows that the model is very accurate in identifying when a loop will not suffer data dependence violations. On average, 79% of the loops are correctly predicted as not suffering violations, and of those that do not suffer violations 97% are

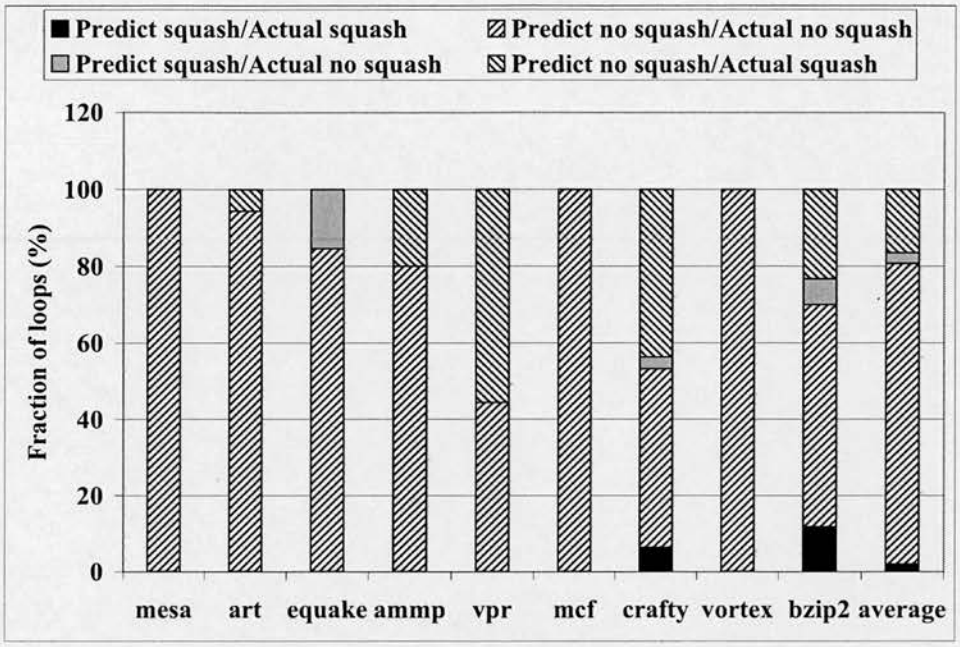

Figure 8.3: Outcome of squash predictions with respect to actual observed squash or no squash. In this experiment all loops except Doall loops are considered.

correctly predicted. This is the case for all the loops identified as not having dependences (Section 6.1.2), which indicates that the data dependence detection mechanisms used in this particular implementation of the model are sufficient. However, many of such correct predictions are for loops identified as having dependences, which indicates that the use of (static) timestamps in the model is enough to correctly model the run-time events. However, from this figure we can see that the model fails to detect a significant fraction of data dependence violations. On average, 16% of the loops are incorrectly predicted as not suffering violations, and of those that do suffer violations only 11% are correctly identified. This is mainly because the current implementation of the model does not take scalar variables into account when looking for possible data dependences.

8.2.3 Prediction Errors

In some cases it may be useful to have an accurate prediction of the actual value of the speedup or slowdown. Figure 8.4 shows the cumulative error distribution for each application plotted at 10% boundaries. The errors are computed as the absolute difference between the estimated speedups and the actual speedups obtained with the simulation, divided by the actual speedup. Thus, in this figure a point (x, y) in the curve means that $y\%$ of the loops have errors less than $x\%$.

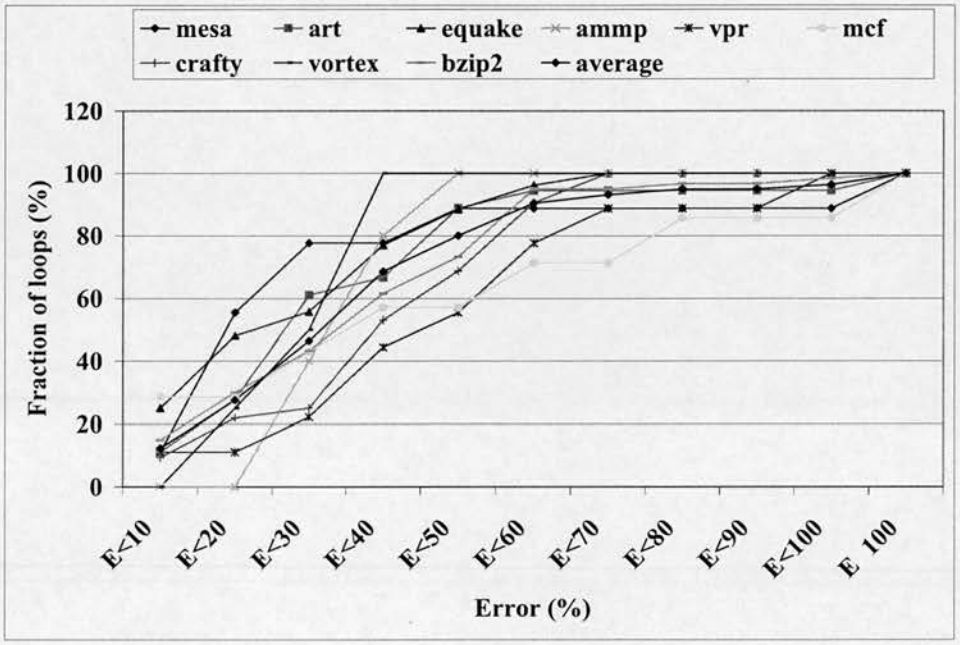

Figure 8.4: Cumulative error distributions. In this experiment all loops except Doall loops are considered.

From this figure we can see that this model is reasonably accurate and is able to predict the actual speedup or slowdown within 20% for 28% of the loops on average and within 50% for 80% of the loops on average.

A closer look at the results shows that the largest errors appear when the model estimates a smaller speedup (or larger slowdown) than what is actually observed. These cases tend to relate to relatively small loops and the reason for this overestimation of the slowdowns can be attributed to differences between the thread sizes estimated from

the SUIF IR and the actual thread sizes. Table 8.1 lists the sources of errors for the top 10% loops with the largest errors (18 loops in total), along with the number of times these sources created such large errors.

Description of source	*Number of instances*	*Range of errors (%)*
Unknown iteration count of inner loop	4	77 to 390
Inaccurate thread size estimation with SUIF	8	57 to 137
Biased conditional	5	58 to 67
Mis-prediction of the squash overhead	2	58 to 62

Table 8.1: Observed sources of prediction errors for the top 10% of loops with the largest errors.(One instance under the category of *mis-prediction of the squash overhead* is also present in the category of *biased conditional*.)

This table shows that the reasons for the very large prediction errors are the lack of information at compile time or the incorrect workload computed from the SUIF intermediate representation.

8.2.4 Comparison with Original Tuple Model

Comparing the results with the extended tuple model and including loops that may suffer data dependence violations against the results with the original tuple model and not including such loops, it can be noted that prediction accuracy has decreased slightly, but not significantly. The errors reported in Section 8.2.3 are larger than the ones with the original model, but are mainly because that there are more outer loops in the evaluation of the extended tuple model, and they are very likely to carry inner loops with statically unknown iteration count. The main effect of this reduction in accuracy is the increase in the fraction of incorrectly predicted speedups at the expense of correctly predicted speedups; and there is little change in the fractions of correctly and incorrectly predicted slowdowns.

8.3 Performance Improvements

Finally, the actual performance impact of using the model to speculatively parallelise only loops that are predicted to have speedups is estimated. Figure 8.5 shows the overall speedups obtained for the execution of all the loops under consideration. In this plot a speedup of one means an execution time equal to the sequential execution time of the loops. This figure shows the result of speculatively parallelising loops following the predictions of the framework (*Selective*) and following a policy to speculatively parallelise all the loops (*Naive*). The figure also shows the result of following a simple selection policy based on a minimum predicted thread size of 50 instructions (*Threshold*). Such policy has been used in previous work (e.g. [49]) as a heuristic to amortise any potential overheads from speculative multithreaded execution. For reference, the figure also shows the result of parallelizing only the Doall loops (*Doall*), assuming perfect speedup for these loops (i.e., 4). Finally, the figure also shows the result of selecting the loops based on a perfect knowledge of the expected speedups and slowdowns (*Oracle*).

From this figure we observe that the selections based on the results of our model perform as well as or better than *Naive* for all applications; and as well as or better than *Threshold* for all applications except *179.art*. The performance gains of our model over these schemes are 6% and 4% on average and as high as 42% and 58%, respectively. Also, the selection policy based on our model is able to curb the large slowdowns with *Naive* in the cases of *177.mesa* and *181.mcf*. This was expected since our model incorrectly predicts speedups in only 31% of the cases on average (Figure 8.2). Finally, our selection policy achieved performance within 4% of that of *Oracle* and was the closest selection policy, on average, to *Oracle*. This was expected since our model incorrectly predicts slowdowns in less than 2% of the cases on average (Figure 8.2).

8.4 Compilation Performance

In closing, in addition to evaluating the quality of the model and the impact it has on the performance of speculative parallelisation, the performance of the model itself is also evaluated. Table 8.2 shows, for each application, the wall-clock execution time of the SUIF1 pass on a 2.8GHz Pentium 4 Xeon machine with 1GBytes of main memory and running Red Hat Linux 3.2.2-5. The table also shows the number of lines of

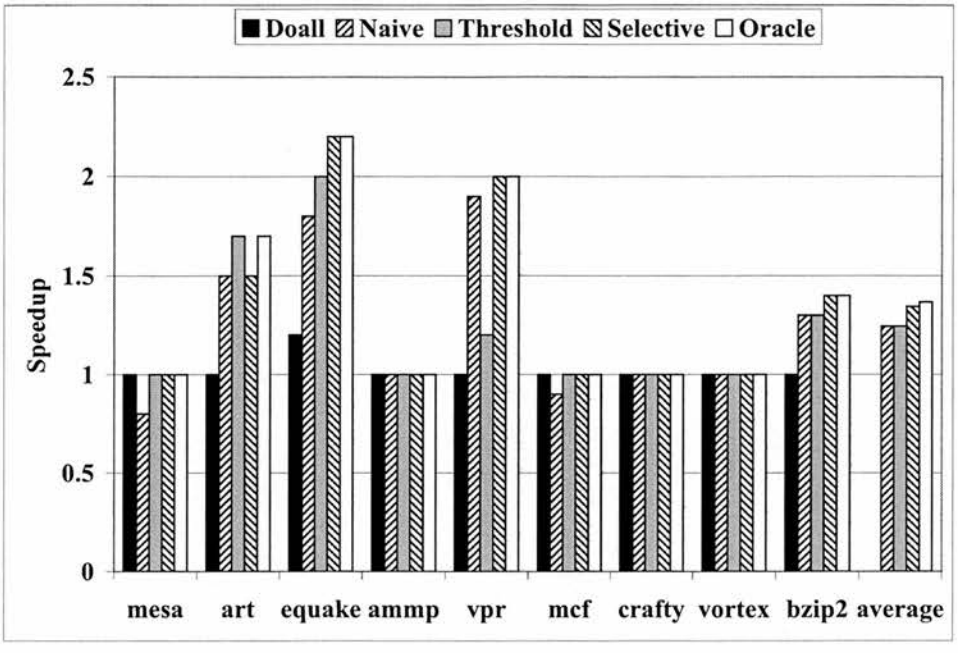

Figure 8.5: Overall performance gains. In this experiment only loops that are neither Doall nor inner loops of Doall or speculatively parallelized loops are considered.

executable code, in thousands, (KLOC) for each application. For comparison the table also shows the wall-clock execution time of gcc 3.2.2 with `-DSPEC_CPU2000 -O3 -fomit-frame-pointer` flags set.

This table shows that except *188.ammp* and *186.crafty*, the execution time of the pass is shorter than the compilation time of gcc O3.

The execution time of the speculative parallelisation cost model is very reasonable in practise because the number of thread sizes generated in the benchmark suite is relatively small. Table 8.3 shows, for each application, the average and maximum sizes of the base and extended thread tables.

This table shows that, except for *256.bzip2*, the number of thread sizes that the model has to manipulate (N in Section 5) is indeed small. It can also be noted that, with the exception of *188.ammp*,*186.crafty*, and *256.bzip2*, the addition of overhead thread sizes does not significantly increase the average and maximum number of thread sizes. This is partly because the model does not incorrectly identify too many violations (Section 8.2.2).

Application	*KLOC*	*Spec. pass exec. time (s)*	*gcc exec. time (s)*
177.mesa	50	1.67	19.8
179.art	1.3	0.08	0.52
183.equake	1.5	0.14	0.65
188.ammp	13	45.57	5.38
175.vpr	17	3.71	5.16
181.mcf	1.9	0.04	0.95
186.crafty	21	25.56	8.93
255.vortex	53	0.62	17.86
256.bzip2	4.6	0.66	1.23

Table 8.2: Performance comparison of speculative parallelisation pass and gcc.

	base thread table size		*Extended thread table size*	
Application	*Average*	*Max.*	*Average*	*Max.*
177.mesa	7	54	7	54
179.art	3	8	3	8
183.equake	1.8	12	1.9	12
188.ammp	7	97	7.8	194
175.vpr	3.2	9	3.2	9
181.mcf	6	16	6	16
186.crafty	4.8	90	6.8	162
255.vortex	1.5	3	1.5	3
256.bzip2	2.8	64	3.8	128

Table 8.3: Average and maximum number of different thread sizes used in the speculative parallelisation cost model.

Chapter 9

Summary of the Simplifications Techniques and Methods

The computational complexity and the complexity of implementation are also primary design constraints of a compiler framework. From the implementer's point of view this limits whether the compiler can be implemented in practice and determines its cost. To the user this limits whether it is computationally viable or compilation is beyond what is tolerable. To make the compiler cost model proposed in this thesis meet such standards, various simplifications were proposed, at the expense of some additional inaccuracy. The simplifications proposed in this thesis can be classified into two categories: *model simplifications* and *implementation simplifications*. Each of them will be discussed in this chapter.

Model simplifications are those that aim to reduce the computational complexity of the cost model. Such simplifications are then intrinsic to the model itself and their elimination would require re-working the model. Implementation simplifications are those that were required in order to keep the implementation of the compiler prototype manageable. Such simplifications are not intrinsic to the model itself and can be reduced, or eliminated, with the incorporation of more elaborate static and/or profile analyses to the compiler framework. However, incorporating such analyses and quantifying their impact on the model's accuracy is beyond the scope of this thesis.

While a detailed quantitative evaluation of the impact of the simplifications to the model and the prototype implementation are not possible, this chapter aims to help the user gain a better understanding of such simplifications and the effect they may

have on the speedup prediction. Each of the simplifications are discussed in detail in the following sections. As some of the simplifications have already been discussed in previous chapters, this chapter also serves as a summary for all the simplifications in the thesis.

9.1 Model Simplifications

9.1.1 Simplification of the Inner Loop Representation using the CCFG

Since the cost model is based on computing the thread sizes derived from all the possible execution paths, a modified version of the CFG, the CCFG, was designed for the program control structure navigation to avoid cycles in the presence of inner loops. The CCFG differs from the CFG in two aspects: firstly, to avoid recomputing the workload during the graph search, each basic block node in the CCFG is annotated with its estimated workload. Secondly, to avoid cycles during program execution path, the backward arcs of the CFG are removed; instead, inner loops are represented by annotating the basic block nodes of inner loops with the estimated inner loop iteration counts. The second difference is a major simplification for the computational complexity in the presence of inner loops and is a trade off of accuracy.

Suppose a CCFG with an inner loop of M iterations, without the second simplification of removing the backward arc, during the program navigation to follow the execution path the depth first search algorithm would repeatedly follow the backward arc $M-1$ times. If the inner loop has control structures that can lead to N possible control paths, then this inner loop would contribute at least N^M execution paths. It would be a dramatic increase on the size of the base thread table and the computational complexity of the model. By using the above simplification technique, cycles on backward arcs are avoided and all of the M iterations are assumed to follow the same execution path for a given execution (different execution instances of the inner loop can follow different paths). This simplification reduces the computational complexity from $O(N^M)$ down to $O(N)$.

While this simplification reduces the computational complexity, it also leads to inaccuracy as some possible execution paths (and, thus, thread sizes) would not be

considered by the cost model. Assume that for a given iteration of the outer loop, the inner loop has N possible execution paths, each with a unique workload, $A = \{W_1, W_2, , , , W_N\}$, where $W_1 \leq W_2 \leq ... \leq W_N$. After the simplification of removing the backward arc, the workload that can be derived from this inner loop is always M times the workloads in A, $W_1 \cdot M \leq W_2 \cdot M \leq ... \leq W_N \cdot M$. The actual N^M threads sizes are a linear combination of the sizes in the set A: they are composed of $W_1 \cdot a_1 \leq W_2 \cdot a_2 \leq ... \leq W_N \cdot a_n$, where $a_1 + a_2 + ... + a_n = N$. It can be observed that these thread sizes fall in the range of $[W_1 \cdot M, W_n \cdot M]$, just as the simplified thread sizes do.

Since the model consider fewer possible thread sizes within the same range of thread sizes, the model is expected to predict more load imbalance than the reality. Another way to see this is to realise that while the model considers no thread size between $W_1 \cdot M$ and $W_2 \cdot M$ there may be up to M thread sizes in between with only slight difference in size. Thus, this simplification is likely to make an under prediction of the speedup compared to the actual value. It can also be noticed that the inaccuracy varies with the number of distinct workload generated by the inner loop, N. So the inaccuracy increases with N. Section 9.2.3 will have more discussion on this simplification.

9.2 Implementation Simplifications

As discussed, the implementation simplifications are implementation specific and are not intrinsic to the cost model. However, they are used in the prototype compiler framework and have bearings on the evaluation of this thesis. In the following sections, each of such simplifications is discussed in detail.

9.2.1 Conditional Branch Probabilities

Since the result of the test statement of the conditional branch can usually only be determined at run-time, the probabilities of the conditional branches are usually very difficult to predict statically. While there are some existing static techniques, such as global value range analysis [35], the tradeoff of incorporating such analysis is the additional complexity of implementation for a limited improvement on the prediction accuracy. The implementation of the cost model can be extended to include such analysis. This can be done by implementing the analysis in a pre-pass and annotate the analysis results to the basic block nodes and then integrating it with the cost model by letting

the cost model read in the results from the basic block nodes during the CCFG navigation. Considering the complexity of implementing a complete global value range analysis, during the design stage of the prototype, the decision was made to trade off the accuracy gained from such additional analysis for the simplicity of implementation by assuming that the branches of conditional statements have equal probabilities on each side. An alternative to such complex static analyses is profiling to obtain the actual control flow probabilities (within variations with different input sets). While the implementation of the cost model can certainly be extended to use such profile information, this goes somewhat against the main philosophy of this thesis of providing a static compile-time solution to the prediction of speculative multithreading performance.

By assuming 50% 50% probabilities for conditional branches, the cost model is likely to mispredict the probabilities of the thread sizes if the conditional branches in the speculative loops are biased. This misprediction has effect on the accuracy of the probabilities of the tuples and also the probabilities of the memory operations on these thread size.

To understand this effect better, let us consider an example speculative loop, with only one conditional statement. Suppose this speculative loop can generate two thread sizes from the two conditional branches, W_1 and W_2, and their probabilities are p_1 and p_2 ($p_2 = 1 - p_1$). Using the equations proposed in Section 7.2, the speedup can be computed as $S = 4 \cdot \frac{W_1 \cdot P_1 + W_2 \cdot (1-P_1)}{W_1 \cdot p_1^4 + W_2 (1-p_1^4)}$. The following plots shows how the predicted speedup (y axis) varies with probability of p_1 (x axis). Figure 9.1 varies the ratio of $W_1 : W_2$. Assuming four processors, so $S \in [0,4]$, and $p_1 \in [0,1]$.

Since the model assumes $p_1 = p_2 = 50\%$, the expected speedup prediction is always the value of y where $x = 0.5$. It can be observed from the plots that a biased conditional ($p_1 = 0\%$ or $p_1 = 100\%$) will lead to an error on the speedup prediction range from 0% to 50%, where 0% is when $W_1 : W_2 = 1$ and 50% is when $W_1 : W_2$ approaches zero. In the following plots, the worst case is when $p_1 = 50\%$ and where $W_1 : W_2 = 1 : 1000$ and the best case is $W_1 = W_2$.

So it can be safe to draw a conclusion that the inaccuracy attributed by assuming 50% for conditional branching probability is affected by the ratio between the two thread sizes and the higher the ratio, the more the inaccuracy. It can also be argued that the error caused by incorrect assumption of branching probability is bounded by 50%.

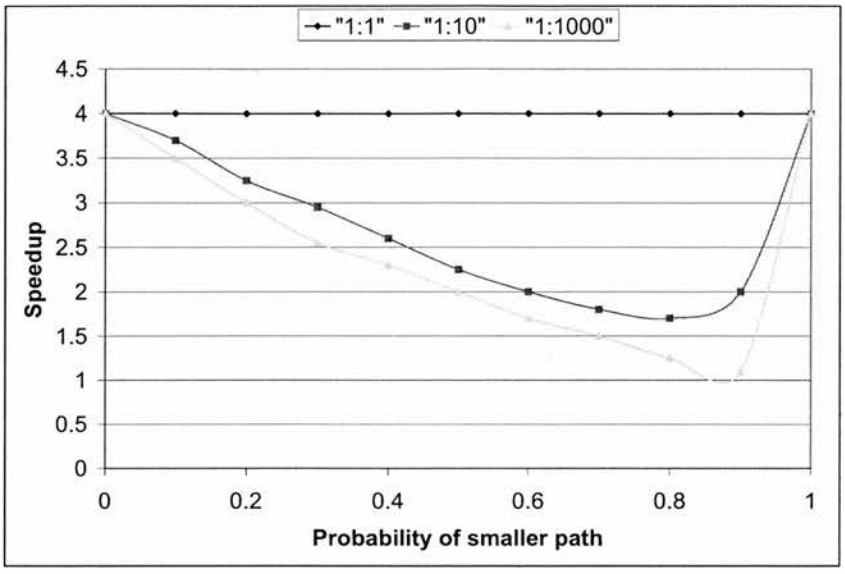

Figure 9.1: The value of the speedup varies with the probability and the ratio of $W_1 : W_2$.

In the experiments, out of the 190 loops studied, 43% of them are only composed of one or more conditional control flow statements, and 14% of them are completely composed of a single conditional statement.

For the 43% loops with conditional statements, 29% of them have an error less than 20% and 79% of them have an error less than 50%. For the 14% loops with only one conditional, 31% of them have an error less than 20% and 86% of them have an error less than 50%.

In contrast, comparing with the loop without any control flow, out of 190 loops, there are 36% of loops without any control flow, in which, 34% of them have an error less than 20% and 91% of them have an error less than 50%. Comparing with the overall average, of all the 190 loops studied, 28% of them have an error of less than 20% and 80% of them have an error of less than 50%.

So it can be observed that compared to the loops without any control flow, there is a decrease in the prediction accuracy for the loops composed of one or more conditional control flow statements, however, compared to the overall average prediction accuracy, the loops composed of one or more conditional control flow statements is still slightly higher.

9.2.2 Prediction of the Inner Loop's Iteration Count

Because the cost model is based on estimating the likely thread sizes of the speculative loop, when the speculative loop has inner loops the cost model needs to know the iteration counts of these inner loops to estimate the thread sizes. If the inner loops' upper and lower bounds are constants, then the iteration count can be easily computed statically. However, when the loops' lower or upper bounds are variables or the return values from function calls, the compiler usually can not determine their iteration counts statically.

In such cases, one possible approach is to use run-time profiling. The implementation of the cost model proposed in this thesis can be easily integrated with profiling feedback. However, the implementation and the evaluation of the cost model with profiling feedback is out of the scope of this thesis and goes somewhat against the main philosophy of this thesis of providing a static compile-time solution to the prediction of speculative multithreading performance.

This thesis used the published results of the static technique of [50], which suggested that five iterations per loop on average can usually be assumed for SPEC benchmarks. So for the simplicity of implementation, when the iteration count cannot be determined statically, the implementation of the cost model assumes that the loop has five iterations. The evaluation in this thesis using the MinneSPEC reduced input set [22] also suggested that five iterations give a reasonable estimate.

Unlike with the static estimation of conditional probabilities the errors due to incorrect estimation of the iteration counts of inner loops is more difficult to bind. This is because the actual iteration count at run time can be any arbitrarily large number.

In the experiments, out of the 190 loops studied, 21% of them have inner loops, in which, 18% have statically unknown inner loop iteration counts and 3% have constant inner loops iteration counts.

For the 18% of loops with statically unknown inner loop iteration counts, 11% of them have an error less than 20% and 62% of them have an error less than 50%.

In contrast, comparing with the loop with constant iteration counts, there are 3% of loops have constant iteration counts, in which, 20% of them have an error less than 20% and 80% of them have an error less than 50%. Comparing with the overall average, of all the 190 loops studied, 28% of them have an error of less than 20% and 80% of them have an error of less than 50%.

So it can be observed that compared to the loops with constant iteration counts, there is a decrease in the prediction accuracy for the loops with statically unknown inner loop iteration counts, and compared to the overall average prediction accuracy, the loops with unknown inner loop iteration counts have a much lower prediction accuracy.

9.2.3 The Combined Error of the Simplifications of Iteration Count and Probabilities of the Conditional Statements

When the speculative loops have inner loops with conditional statements inside, or a conditional statement with a loop on one of its branches, the errors due to the simplifications are combined, which leads to a further prediction error.

The case that the nested loop has conditional statement inside was already discussed in section 9.1.1. When the speculative loop has a conditional statement with a nested loop on one of its branches, this loop is likely to have a high load imbalance overhead if it is not biased, since the branch with the nested loop usually has a much higher thread size than another branch.

In the experiments, out of the 190 loops studied, there are 17% of them which carry inner loops and have at least one conditional statements and there are 4% of them with only have nested loops but without any conditional.

For the 17% of loops which carry inner loops and have at least one conditional statements, 13% of them have an error less than 20% and 59% the of them have an error less than 50%.

In contrast, comparing with the loop with nested loops but without any conditional, there are 4% of loops have nested loops but without any conditional, in which, 20% of them have an error less than 20% and 75% of them have an error less than 50%. Also comparing with the loops with only conditionals, in which, 29% of them have an error less than 20% and 79% of them have an error less than 50%. Again, comparing with the overall average, of all the 190 loops studied, 28% of them have an error of less than 20% and 80% of them have an error of less than 50%.

So it can be observed that compared to the loops with nested loops but without any conditional, and comparing to the loops with only conditionals but without nested loops, there is a decrease in the prediction accuracy for the loops with nested loops and with at least one conditional, and compared to the overall average prediction accuracy, the loops with nested loops and at least one conditional statement have a much lower

prediction accuracy.

9.2.4 Data Dependence prediction

Date dependence prediction is used for predicting the squash and restart overhead thread sizes. The prediction of a data dependence requires the knowledge of the addresses and the time stamps (cycle numbers relative to the beginning of the thread) of the memory access operations. However, when determining whether two memory operations are referring to the same address, there is no need to determine the exact addresses of the memory accesses, since it is sufficient to compare the two memory accesses by their variable symbols, if they are in the same life scope.

If the variables are array accesses, they can be compared by the symbols of their base addresses, and their array indexes. When the array index a variable, particularly, a loop induction variable, there are some existing techniques to detect the loop carried array dependences, such as GCD test (greatest common divisor) and the Banerjee-Wolfe test [3]. These tests can be integrated into the cost model for predicting squash and and restart overhead. As implementing these tests increases the implementation complexity of the compiler prototype, for the simplicity of implementation, these tests were not implemented in the evaluation framework. In the experiments, when two array accesses have statically unknown array indexes they are conservatively assumed to refer to the same address. The time stamps of the memory operations are statically estimated by counting the SUIF intermediate instructions with each instruction weighted by its estimated cycle count on the target architecture.

Observing the experiment results, the prediction accuracy of the data dependence varies from different benchmarks. In the regular applications with regular array or vector accesses, the data dependence tends to be over-estimated, since that array indexing is not taken into account.

Observing the experimental results in figure 8.3, on average, there are 2% of the loops are predicted squash but actual no squash. Among these benchmarks, equake has the most mis-prediction of 15% attributed to this reason since there are most array accesses in equake. Another two benchmarks that have prediction errors attributed to this reason are bzip2 with 7% and crafty with 2% these are also expected since bzip2 and crafty have many array accesses.

9.2.5 Speculative Buffer Overflow Estimation

To estimate the speculative buffer overflow overhead for a thread the compiler needs to know the amount of speculative data generated by this thread and the capacity of the speculative buffer. The capacity of the speculative buffer is a combined factor of the size of the buffer and its associativity. This usually varies across different speculative multithreading architecture configurations. Given the same speculative buffer size, the probability of the speculative buffer overflow also varies with the associativity of the speculative buffer. There are three kinds of associativities: fully associative, direct mapped and set associative. Each of them is discussed in the following sections.

9.2.5.1 Directed Mapped

In the directed mapped cache, the mappings to the cache lines are usually determined by the lest significant bits of the target address. Suppose the cache line size is 2^n, then the bottom n bits of the address correspond to an offset within a cache entry and if the cache can hold 2^m entries, then the next m address bits give the cache location. The remaining top bits of the address are stored as tags along with the cache entry.

With a direct mapped speculative buffer, the speculative thread stalls when a store operation is mapped to a cache line which already holds the speculatively modified values (as these cannot be written back to the shared storage). So predicting the speculative buffer overflow for the direct mapped speculative buffers requires some knowledge of the target addresses. Without the exact addresses, the behaviour in the direct mapped buffer can sometimes be predicted using cache miss equations [21] for regular applications. However, in general, determining the addresses of the memory access at compile time is a very difficult problem, due to the lack of run-time information, which in turn, makes predicting the speculative buffer overflow overhead on a direct mapped speculative buffer difficult.

Though the exact behaviour cannot be determined, it can be observed that with the number of cache lines filled into the buffer, the probability of overflow, p_{ovflow} increases. The worst case is that two adjacent store operations conflict with each other and, thus, on the second store operation the speculative thread cannot proceed further until the previous speculative value has been committed to the memory. On the other hand, the best case is that there is no conflict between the store operations, so the speculative threads keeps buffering speculative values into the speculative buffer until

it is full. This behaviour can be captured in the cost model, by adjusting p_{ovflow} for each store operation. Determining the probability of stall (p_{ovflow}) for direct mapped buffer is out of the scope of this thesis.

9.2.5.2 Fully Associative

In a fully associative cache, data from any address can be stored in any cache location. The whole data address, except the byte offset, is used as tag. So using the fully associate cache solves the problem of conflict for cache locations since the speculative thread only stalls when the whole cache is full. A typical proposal that uses fully associative buffer is the Standford Hydra CMP.

Based on the size of the fully associative speculative buffer, the compiler can estimate whether the speculative thread stalls the buffer by comparing the amount of the speculative data it generates against the size of the speculative buffer. Assuming that distinct variables are stored in different and unique memory locations, then the problem of determining the store operation that stalls the speculative buffer is narrowed down to simply counting the number of store operations to distinct variable symbols.

An assumption made is that distinct variables store to different memory addresses. This can not be estimated, however, if the variable is a array, with variable indexes. In such cases, the assumption was made that each array operation will store to a different location. This is also a simplification to trade off some accuracy for a complicated array subscript analysis. If the assumption is wrong, and the different array store operations store to the same address, the effect is that the cost model will count more store operations than there actually are. So the model tends to predict more speculative buffer overhead. The amount of over-prediction is determined by the number of mispredicted array store operations. For the simplicity of implementation, the compiler model, proposed in this thesis assumes using a fully associative speculative buffer like the Hydra CMP and in the experiment, no loop overflows the speculative buffer.

9.2.5.3 Set Associative

Set associative cache is a compromise between direct mapped cache and fully associative cache where each address is mapped to a certain set of cache locations. With an n-way set associative speculative buffer with m sets, it has n cache locations in each set. A cache line, if mapped to a set, may be stored in any of the n locations in that set,

with its upper address bits as a tag. Thus, the speculative thread stalls if any of the n sets reach its full capacity.

Similar to the direct mapped speculative buffer, determining which set a store operation is going to be mapped to is also determined by part of its address, which is usually a certain number bits, $(log(n))$, taken from the middle of the address. This is difficult to predict at compile time, due to the lack of information,

Similar with the approach with the direct mapped cache, this can be fitted into the cost model by adjusting p_{ovflow} for each store operation. The worst case is that all the store operations conflict, thus the number of store operations that the speculative buffer can accommodate is equal to the associativity n of the cache. On the other hand, in the optimal case, the capacity is equal to the number of cache lines. Again, determining the probability of stall (p_{ovflow}) for set associative speculative buffer is out of the scope of this thesis.

Part IV

Future Work and Conclusion

Chapter 10

Future Work

This thesis has made a first attempt at a model-driven static compiler framework for speculative multithreaded execution performance prediction. This framework is flexible to include all major speculative multithreading overheads and it is open to be extended in many ways.

Possible extensions include a further study of the probability of dependence, p_{dep} in Equation 7.8. Such study would likely involve incorporating some memory disambiguation analysis techniques, such as alias analysis or subscripted subscript analysis.

Parallel to p_{dep} is the probability of overflow, p_{ovflow} in Equation 7.9. A further study of the overflow probability would likely involve the behaviour of the speculative buffer under different associativities.

Finally, the experimental result shows that the prediction accuracy can be improved if biased conditionals can be detected and statically unknown inner loop iteration count can be better estimated. Such analysis would involve some techniques such as symbolic analysis or profiling if run time feedback is feasible. Such technique can also be applied to the analysis of the iteration count for the inner loop.

Chapter 11

Conclusion

This thesis proposed and evaluated a model of speculative multithreaded execution that can be used by the compiler to reason about the overheads and expected resulting performance gains, or losses, from speculative parallelisation. This model is based on estimating the likely combined run-time effects of various speculative parallelisation overheads, and properly takes into account the scheduling restrictions of most speculative execution environments. The model is based on estimating the likely execution duration of threads and considers all the possible permutations of these threads when scheduled on a multiprocessor. The model is flexible enough to include all speculative parallelisation overheads. Different from previous work which present heuristics that attempt to estimate "good" or "bad" sections for speculative multithreaded execution, this proposed compiler framework attempts to quantitatively estimate the speedup or slowdown. Such estimate can then be used by the compiler or run-time system to make more complex and educated tradeoff decisions. For instance, in a highly loaded multiprogrammed environment the compiler or run-time system may decide to switch off speculative parallelisation even when a speedup is expected if this speedup is too small and does not justify the use of the extra resources.

The model proposed requires data structures that are simple to generate and manage and formulas that are fast to compute. Also, where compile-time information is too incomplete, the accuracy of the model could be improved with only simple profile information that can be obtained from a sequential execution of the program. Finally, the model can be easily added to existing compiler frameworks and requires little modification to common intermediate representations.

This compiler model was implemented in a research compiler development frame-

work. The model seamlessly uses the compiler's intermediate representation and integrates with the control and data flow analyses. The resulting framework was tested and evaluated on a collection of SPEC benchmarks, which include large real-world scientific and engineering applications. The framework was found to be very stable and efficient with moderate compilation times.

Initially, the proposed framework is evaluated on a number of loops that suffer mainly from load imbalance and thread dispatch and commit overheads. Experimental results show that the framework can identify on average 68% of the loops that cause slowdowns and on average 97% of the loops that lead to speedups. In fact, the framework predicts the speedups or slowdowns with an error of less than 20% for an average of 44% of the loops across the benchmarks, and with an error of less than 50% for an average of 84% of the loops. Overall, the framework leads to a performance improvement of 5% on average, and as high as 38%, over a naive approach that attempts to speculatively parallelize all the loops considered.

The proposed framework is then evaluated on loops that may suffer from data dependence violations. Experimental results with all loops show that prediction accuracy is lower when loops with violations are included. Nevertheless, accuracy is still very high for a static model: the framework can identify on average 45% of the loops that cause slowdowns and on average 96% of the loops that lead to speedups; it predicts the speedups or slowdowns with an error of less than 20% for an average of 28% of the loops across the benchmarks, and with an error of less than 50% for an average of 80% of the loops. Overall, the framework often outperforms, by as much as 25%, a naive approach that attempts to speculatively parallelize all the loops considered, and is able to curb the large slowdowns caused in many cases by this naive approach.

Bibliography

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. "Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures." *Intl. Symp. on Computer Architecture*, pages 248-259, June 2000.

[2] H. Akkary and M. A. Driscoll. "A Dynamic Multithreading Processor." *Intl. Symp. on Microarchitecture*, pages 226-236, December 1998.

[3] U. Benerjee and M. A. Driscoll. "Dependence Analysis for Supercomputing" Kluwer Academic Publishers, Norwell, MA, 1988.

[4] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. "Advanced Program Restructuring for High-Performance Computers with Polaris." *IEEE Computer*, Vol. 29, No. 12, pages 78-82, December 1996.

[5] A. Bhowmik and M. Franklin. "A General Compiler Framework for Speculative Multithreading." *Symp. on Parallel Algorithms and Architectures*, pages 99-108, August 2002.

[6] S. Chatterjee, E. Parker, P. J. Hanlon and A. R. Lebeck. "Exact Analysis of the Cache Behavior of Nested Loops." *Conf. on Programming Language Design and Implementation*, pages 286-297, June 2001.

[7] M. Chen and K. Olukotun. "TEST: a Tracer for Extracting Speculative Threads." *Intl. Symp. on Code Generation and Optimization*, pages 301-312, March 2003.

[8] P.-S. Chen, M.-Y. Hung, Y.-S. Hwang, R. D.-C. Ju, and J. K. Lee. "Compiler Support for Speculative Multithreading Architecture with Probabilistic Points-to Analysis." *Symp. on Principles and Practice of Parallel Programming*, pages 25-36, June 2003.

[9] M. Cintra, J. F. Martínez, and J. Torrellas. "Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors." *Intl. Symp. on Computer Architecture*, pages 13-24, June 2000.

[10] M. Cintra and J. Torrellas. "Eliminating Squashes Through Learning Cross-Thread Violations in Speculative Parallelization for Multiprocessors." *Intl. Symp. on High-Performance Computer Architecture*, pages 43-54, February 2002.

[11] M. Cintra and D. R. Llanos. "Toward Efficient and Robust Software Speculative Parallelization in Multiprocessors." *Symp. on Principles and Practice of Parallel Programming*, pages 13-24, June 2003.

[12] F. Dang, H. Yu, and L. Rauchwerger. "The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops." *Intl. Parallel and Distributed Processing Symp.*, pages 20-29, April 2002.

[13] J. Dou and M. Cintra. "Compiler Estimation of Load Imbalance Overhead in Speculative Parallelization." *Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 203-214, September 2004.

[14] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai. "A Cost-Driven Compilation Framework for Speculative Parallelization of Sequential Programs." *Conf. on Programming Language Design and Implementation*, pages 71-81, June 2004.

[15] P. Dubey, K. O'Brien, K. M. O'Brien, and C. Barton. "Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-assisted Fine-Grained Multithreading." *Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 109-121, June 1995.

[16] M. J. Garzarán, M. Prvulovic, J. M. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas. "Tradeoffs in Buffering Memory State for Thread-Level Speculation in Multiprocessors." *Intl. Symp. on High-Performance Computer Architecture*, pages 191- 202, February 2003.

[17] V. Govindan and M. A. Franklin. "Application Load Imbalance on Parallel Processors." *Intl. Parallel Processing Symp.*, pages 836-842, April 1996.

[18] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S-W. Liao, E. Bugnion, and M. Lam. "Maximizing Multiprocessor Performance with the SUIF Compiler." *IEEE Computer*, Vol. 29, No. 12, pages 84-89, December 1996.

[19] L. Hammond, M. Wiley, and K. Olukotun. "Data Speculation Support for a Chip Multiprocessor." *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 58-69, October 1998.

[20] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Chen, K. Olukotun. "The Stanford Hydra CMP." *IEEE Micro*, Vol. 12, No. 2, pages 71-84, March-April 2000.

[21] S. W. Kim and R. Eigenmann. "The Structure of a Compiler for Explicit and Implicit Parallelism." *Intl. Wksp. on Languages and Compilers for Parallel Computing*, August 2001.

[22] A. J. KleinOsowski and D. J. Lilja. "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research." *Computer Architecture Letters*, Vol. 1, June 2002.

[23] V. Krishnan and J. Torrellas. "A Chip-Multiprocessor Architecture with Speculative Multithreading." *IEEE Trans. on Computers, Special Issue on Multithreaded Architectures*, Vol. 48, No. 9, pages 866-880, September 1999.

[24] Z. Li, J.-Y. Tsai, X. Wang, P.-C. Yew, and B. Zheng. "Compiler Techniques for Concurrent Multithreading with Hardware Speculation Support." *Intl. Wksp. on Languages and Compilers for Parallel Computing*, August 1996.

[25] W. Liu, J. Tuck, L. Ceze, K. Strauss, J. Renau, and J. Torrellas. "POSH: A Profiler-Enhanced TLS Compiler that Leverages Program Structure." *Watson Conf. on Interaction between Architecture, Circuits, and Compilers*, September 2005.

[26] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, B. Werner. "Simics: A Full System Simulation Platform." *IEEE Computer*, Vol. 35, No. 2, pages 50-58, February 2002.

[27] P. Marcuello, A. González, and J. Tubella. "Speculative Multithreaded Processors." *Intl. Conf. on Supercomputing*, pages 77-84, July 1998.

[28] P. Marcuello and A. González. "Clustered Speculative Multithreaded Processors." *Intl. Conf. on Supercomputing*, pages 365-372, June 1999.

[29] P. Marcuello and A. González. "Thread-Spawning Schemes for Speculative Multithreading." *Intl. Symp. on High-Performance Computer Architecture*, pages 55-64, February 2002.

[30] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. "Dynamic Speculation and Synchronization of Data Dependences." *Intl. Symp. on Computer Architecture*, pages 181-193, June 1997.

[31] K. Olukotun, L. Hammond, and M. Willey. "Improving the Performance of Speculatively Parallel Applications on the Hydra CMP." *Intl. Conf. on Supercomputing*, pages 21-30, June 1999.

[32] C.-L. Ooi, S. W. Kim, I. Park, R. Eigenmann, B. Falsafi, and T. N. Vijaykumar. "Multiplex: Unifying Conventional and Speculative Thread-Level Parallelism on a Chip Multiprocessor." *Intl. Conf. on Supercomputing*, pages 368-380, June 2001.

[33] J. Oplinger, D. Heine, and M. Lam. "In Search of Speculative Thread-level Parallelism." *Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 303-313, October 1999.

[34] S. Palacharla, N. P. Jouppi, and J. E. Smith. "Complexity-Effective Superscalar Processors." *Intl. Symp. on Computer Architecture*, pages 206-218, June 1997.

[35] J. R. C. Patterson. "Accurate Static Branch Prediction by Value Range Propagation." *Conf. on Programming Language Design and Implementation*, pages 67-78, January 1995.

[36] M. K. Prabhu and K. Olukotun. "Using Thread-Level Speculation to Simplify Manual Parallelization." *Symp. on Principles and Practice of Parallel Programming*, pages 1-12, June 2003.

[37] M. Prvulovic, M. J. Garzarán, L. Rauchwerger, and J. Torrellas. "Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization." *Intl. Symp. on Computer Architecture*, pages 204-215, June 2001.

[38] C. G. Quinones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. "Mitosis Compiler: An Infrastructure for Speculative Threading Based on Pre-Computation Slices." *Conf. on Programming Language Design and Implementation*, pages 269-279, June 2005.

[39] L. Rauchwerger and D. Padua. "The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization." *Conf. on Programming Language Design and Implementation*, pages 218-232, June 1995.

[40] R. Sakellariou and J. R. Gurd. "Compile-Time Minimisation of Load Imbalance in Loop Nests." *Intl. Conf. on Supercomputing*, pages 277-284, July 1997.

[41] G. Sohi, S. Breach, and T. N. Vijaykumar. "Multiscalar Processors." *Intl. Symp. on Computer Architecture*, pages 414-425, June 1995.

[42] J. G. Steffan and T. C. Mowry. "The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization." *Intl. Symp. on High-Performance Computer Architecture*, pages 2-13, February 1998.

[43] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. "A Scalable Approach to Thread-Level Speculation." *Intl. Symp. on Computer Architecture*, pages 1-12, June 2000.

[44] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. "Improving Value Communication for Thread-Level Speculation." *Intl. Symp. on High-Performance Computer Architecture*, pages 65-75, February 2002.

[45] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. "The STAMPede Approach to Thread-Level Speculation." *ACM Trans. on Computer Systems*, Vol. 23, No. 3, pages 253-300, August 2005.

[46] J.-Y. Tsai, J. Huang, C. Amlo, D. Lilja, and P.-C. Yew. "The Superthreaded Processor Architecture." *IEEE Trans. on Computers, Special Issue on Multithreaded Architectures*, Vol. 48, No. 9, pages 881-902, September 1999.

[47] X. Vera and J. Xue. "Let's Study Whole-Program Cache Behavior Analitically." *Intl. Symp. on High-Performance Computer Architecture*, pages 176-185, February 2002.

[48] T. N. Vijaykumar. "Compiling for the Multiscalar Architecture." Ph.D. thesis, Department of Computer Science, University of Wisconsin at Madison, January 1998.

[49] T. N. Vijaykumar and Gurindar S. Sohi. "Task Selection for a Multiscalar Processor." *Intl. Symp. on Microarchitecture*, pages 81-92, December 1998.

[50] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison. "Accurate Static Estimators for Program Optimization." *Conf. on Programming Language Design and Implementation*, pages 85-96, June 1994.

[51] F. Warg and P. Stenström. "Improving Speculative Thread-Level Parallelism Through Module Run-Length Prediction." *Intl. Parallel and Distributed Processing Symp.*, pages 12-19, April 2003.

[52] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. "Compiler Optimization of Scalar Value Communication Between Speculative Threads." *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 171-183, October 2002.

[53] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. "Compiler Optimization of Memory-Resident Value Communication Between Speculative Threads." *Intl. Symp. on Code Generation and Optimization*, pages 39-50, March 2004.

[54] Y. Zhang, L. Rauchwerger, and J. Torrellas. "Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors." *Intl. Symp. on High-Performance Computer Architecture*, pages 161-173, February 1998.